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 Vector is array (1 .. 12) of Some_Type;

K : Integer range 0 .. 12 := 0; -- anonymous subtype
Values : Vector;
...
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 Vector is array (Index) of Some_Type;

K : Counter := 0;
Values : Vector;
...
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 Vector is array (Index) of Float;
K : Index;
Values : Vector;
...
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;
```

- Approach 3: use GNAT's intrinsic
  - Conditions on function name and type representation
  - See GNAT UG 8.11

```
function Shift_Left
  (Value  : T;
  Amount : Natural) return T with Import, Convention => Intrinsi
```

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
    - Ada 2022 attributes `T'Enum_Rep`, `T'Enum_Val`

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

```ada
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) = 0`
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) = 0`*
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, A, B, 'C');
for T1 use (NUL => 0, A => 1, B => 2, 'C' => 3);
type T2 is array (Positive range <>) of T1;
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) is(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, A, B, 'C');
for T1 use (NUL => 0, A => 1, B => 2, 'C' => 3);
type T2 is array (Positive range <>) of T1;
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) is(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

Characteristics of Components

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

```
type Record_1 is record
   This_Is_Not_Legal : array (1 .. 3) of Integer;
end record;
```

- **No** constant components

```
type Record_2 is record
   This_Is_Not_Legal : constant Integer := 123;
end record;
```

- **No** recursive definitions

```
type Record_3 is record
   This_Is_Not_Legal : Record_3;
end record;
```

- **No** indefinite types

```
type Record_5 is record
   This_Is_Not_Legal : String;
   But_This_Is_Legal : String (1 .. 10);
end record;
```

Multiple Declarations

- Multiple declarations are allowed (like objects)

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

- Equivalent to

```
type Several is record
   A : Integer := F;
   B : Integer := F;
   C : Integer := F;
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

```
type Record_T is record
   -- Definition here
end record;
```

Which record definition is legal?

A. `Component_1 : array (1 .. 3) of Boolean`
B. `Component_2, Component_3 : Integer`
C. `Component_1 : Record_T`
D. `Component_1 : constant Integer := 123`

Quiz

```
type Record_T is record
   -- Definition here
end record;
```

Which record definition is legal?

A. `Component_1 : array (1 .. 3) of Boolean`
B. *`Component_2, Component_3 : Integer`*
C. `Component_1 : Record_T`
D. `Component_1 : constant Integer := 123`

A. Anonymous types not allowed
B. Correct
C. No recursive definition
D. No constant component

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

Available Operations

- Predefined
  - Equality (and thus inequality)

    ```
    if A = B then
    ```

  - Assignment

    ```
    A := B;
    ```

- 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;
```

Limited Types - Quick Intro

- A **record** type can be limited
  - And some other types, described later
- *limited* types cannot be **copied** or **compared**
  - As a result then cannot be assigned
  - May still be modified component-wise

```
type Lim is limited record
    A, B : Integer;
end record;

L1, L2 : Lim := Create_Lim (1, 2); -- Initial value OK

L1 := L2; -- Illegal
if L1 /= L2 then -- Illegal
[...]
```

Aggregates

Aggregates

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

```
(Pos_1_Value,
 Pos_2_Value,
 Component_3 => Pos_3_Value,
 Component_4 => <>, -- Default value (Ada 2005)
 others => Remaining_Value)
```

Record Aggregate Examples

```
type Color_T is (Red);
type Car_T is record
   Color    : Color_T;
   Plate_No : String (1 .. 6);
   Year     : Natural;
end record;
type Complex_T is record
   Real      : Float;
   Imaginary : Float;
end record;

declare
   Car   : Car_T     := (Red, "ABC123", Year => 2_022);
   Phase : Complex_T := (1.2, 3.4);
begin
   Phase := (Real => 5.6, Imaginary => 7.8);
end;
```

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 : Float;
   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 building and running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer;
   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 building and running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer;
   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 is incomplete. The aggregate must specify all components. You could use box notation `(A => 1, others => <>)`

Quiz

What is the result of building and running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer;
      D : My_Integer;
   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 building and running this code?

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

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

Which assignment(s) is(are) **not** legal?

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

Which assignment(s) is(are) **not** legal?

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

Component Default Values

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

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 2005

```
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 2005

```
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 **is related to** discriminant
- Aggregate assignment is allowed

Discriminants

```
2  type Person_Group is (Student, Faculty);
3  type Person (Group : Person_Group) is record
4     Age : Positive;
5     case Group is
6        when Student => -- 1st variant
7           Gpa  : Float range 0.0 .. 4.0;
8        when Faculty => -- 2nd variant
9           Pubs : Positive;
10    end case;
11 end record;
```

- Group (on line 3) 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**
- Discriminant can be used in *variant part* (line 5)
  - Similar to case statements (all values must be covered)
  - Fields listed will only be visible if choice matches discriminant
  - Field names need to be unique (even across discriminants)
  - Variant part must be end of record (hence only one variant part allowed)

Semantics

- Person objects are **constrained** by their discriminant
  - They are **indefinite**
  - **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,
                  Age => 21,
                  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
  - Type is now **definite**
- 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, Age => 22, Gpa => 0.0);
S := (Group => Faculty, Age => 35, Pubs => 10);
```

Quiz

```
type T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;
```

```
O : T (1);
```

Which component does `O` contain?

A. `O.I, O.B`
B. `O.N`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;
```

```
O : T (1);
```

Which component does `O` contain?

A. `O.I, O.B`
B. *`O.N`*
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Integer) is record
    case Floating is
        when 0 =>
            I : Integer;
        when 1 =>
            F : Float;
    end case;
end record;

O : T (1);
```

Which component does `O` contain?

A. `O.F, O.I`
B. `O.F`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Integer) is record
    case Floating is
        when 0 =>
            I : Integer;
        when 1 =>
            F : Float;
    end case;
end record;

O : T (1);
```

Which component does `O` contain?

A. `O.F, O.I`
B. `O.F`
C. ***None: Compilation error***
D. None: Runtime error

The variant `case` must cover all the possible values of `Integer`.

Quiz

```
type T (Floating : Boolean) is record
    case Floating is
        when False =>
            I : Integer;
        when True =>
            F : Float;
    end case;
    I2 : Integer;
end record;

O : T (True);
```

Which component does `O` contain?

A. `O.F, O.I2`
B. `O.F`
C. None: Compilation error
D. None: Runtime error

Quiz

```
type T (Floating : Boolean) is record
    case Floating is
        when False =>
            I : Integer;
        when True =>
            F : Float;
    end case;
    I2 : Integer;
end record;
```

```
O : T (True);
```

Which component does `O` contain?

A. `O.F, O.I2`
B. `O.F`
C. ***None: Compilation error***
D. None: Runtime error

The variant part cannot be followed by a component declaration (`I2 : Integer` there)

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

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 **is related to** discriminant
- Aggregate assignment is allowed

Example Discriminated 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

```
enum person_group {Student, Faculty};

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

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

Example Defined in Ada

```
type Person_Group is (Student, Faculty);
type Person (Group : Person_Group) is -- Group is the discriminan
  record
    Name : String(1..10); -- Always present
    case Group 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;
```

- Group value enforces component availability
  - Can only access `GPA` and `Year` when `Group` is `Student`
  - Can only access `Pubs` when `Group` 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

Discriminated Record Semantics

Discriminant in Ada Discriminated Records

- Variant record type contains a special *discriminant* component
  - Value indicates which *variant* is present
- When a component 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 (`Group`) established when object of type Person created
  - Run-time check verifies that component selected from variant is consistent with discriminant value
    - Constraint_Error raised if the check fails
- Can only read discriminant (as any other component), 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 := (Group  => 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_Group is (Student, Faculty);
   type Person (Group : Person_Group) is
      record
         Name : String(1..10);
         case Group 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("Group:" & Person_Group'Image(Item.Group));
  Put_Line("Name: " & Item.Name);
  -- Group specified by caller
  case Item.Group 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
  -- Group specified by caller
  case Item.Group 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
  Group   : Person_Group;
  Line  : String(1..80);
  Index : Natural;
begin
  loop
    Put("Group (Student or Faculty, empty line to quit): ");
    Get_Line(Line, Index);
    exit when Index=0;
    Group := Person_Group'Value(Line(1..Index));
    declare
      Someone : Person(Group);
    begin
      Get(Someone);
      case Someone.Group is
        when Student => Student_Do_Something (Someone);
        when Faculty => Faculty_Do_Something (Someone);
      end case;
      Put(Someone);
    end;
  end loop;
end Person_Test;
```

Unconstrained Discriminated Records

Adding Flexibility to Discriminated Records

- Previously, declaration of `Person` implies that object, once created, is always constrained by initial value of `Group`
  - 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 Discriminated Record Example

```
declare
   type Mutant(Group : Person_Group := Faculty) is
      record
         Name : String(1..10);
         case Group 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.Group = Faculty)

begin
   Zork       := Pat;     -- OK, Zork.Group was Faculty, is now Student
   Zork.Group := Faculty; -- Illegal to assign to discriminant
   Zork       := Doc;     -- OK, Zork.Group 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;
   -- V and V2 declaration...
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;
   -- V and V2 declaration...
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

Unconstrained 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 Unconstrained 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 Discriminated Records

- Discriminant can serve as bound of array component

```
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

Varying Length Array via Discriminated Records and Subtypes

- Discriminant can serve as bound of array component
- Subtype serves as upper bound for `Size_T'Last`

```
subtype VString_Size is Natural range 0 .. Max_Length;

type VString (Size : VString_Size := 0) is
  record
    Data   : String (1 .. Size) := (others => ' ');
  end record;

Empty_VString : constant VString := (0, "");

function Make (S : String) return VString is
   ((Size => S'Length, Data => S));
```

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

```ada
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.

Discriminated 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)

```ada
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 Discriminated 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 components common to a set of variants

Type Derivation

Introduction

Type Derivation

- Type *derivation* allows for reusing code
- Type can be **derived** from a **base type**
- Base type can be substituted by the derived type
- Subprograms defined on the base type are **inherited** on derived type
- This is **not** OOP in Ada
  - Tagged derivation **is** OOP in Ada

Ada Mechanisms for Type Inheritance

- *Primitive* operations on types
  - Standard operations like + and -
  - Any operation that acts on the type
- Type derivation
  - Define types from other types that can add limitations
  - Can add operations to the type
- Tagged derivation
  - **This** is OOP in Ada
  - Seen in other chapter

Primitives

Primitive Operations

- A type is characterized by two elements
  - Its data structure
  - The set of operations that applies to it
- The operations are called **primitive operations** in Ada

```
type T is new Integer;
procedure Attrib_Function(Value : T);
```

General Rule For a Primitive

- Primitives are subprograms
- **S** is a primitive of type **T** iff
  - **S** is declared in the scope of **T**
  - **S** "uses" type **T**
    - As a parameter
    - As its return type (for `function`)
  - **S** is above *freeze-point*
- Rule of thumb
  - Primitives must be declared **right after** the type itself
  - In a scope, declare at most a **single** type with primitives

```
package P is
   type T is range 1 .. 10;
   procedure P1 (V : T);
   procedure P2 (V1 : Integer; V2 : T);
   function F return T;
end P;
```

Simple Derivation

Simple Type Derivation

- Any type (except `tagged`) can be derived

```
type Child is new Parent;
```

- Child inherits from:
  - The data **representation** of the parent
  - The **primitives** of the parent
- Conversions are possible from child to parent

```
type Parent is range 1 .. 10;
procedure Prim (V : Parent);
type Child is new Parent;  -- Freeze Parent
procedure Not_A_Primitive (V : Parent);
C : Child;
...
Prim (C);  -- Implicitly declared
Not_A_Primitive (Parent (C));
```

Simple Derivation and Type Structure

- The type "structure" can not change
  - `array` cannot become `record`
  - Integers cannot become floats
- But can be **constrained** further
- Scalar ranges can be reduced

```
type Tiny_Int is range -100 .. 100;
type Tiny_Positive is new Tiny_Int range 1 .. 100;
```

- Unconstrained types can be constrained

```
type Arr is array (Integer range <>) of Integer;
type Ten_Elem_Arr is new Arr (1 .. 10);
type Rec (Size : Integer) is record
   Elem : Arr (1 .. Size);
end record;
type Ten_Elem_Rec is new Rec (10);
```

Overriding Indications

Ada 2005

- **Optional** indications
- Checked by compiler

```
type Root is range 1 .. 100;
procedure Prim (V : Root);
type Child is new Root;
```

- **Replacing** a primitive: `overriding` indication

```
overriding procedure Prim (V : Child);
```

- **Adding** a primitive: `not overriding` indication

```
not overriding procedure Prim2 (V : Child);
```

- **Removing** a primitive: `overriding` as `abstract`

```
overriding procedure Prim (V : Child) is abstract;
```

Quiz

```
type T1 is range 1 .. 100;
procedure Proc_A (X : in out T1);

type T2 is new T1 range 2 .. 99;
procedure Proc_B (X : in out T1);
procedure Proc_B (X : in out T2);

-- Other scope
procedure Proc_C (X : in out T2);

type T3 is new T2 range 3 .. 98;

procedure Proc_C (X : in out T3);
```

Which are T1's primitives

A. Proc_A
B. Proc_B
C. Proc_C
D. No primitives of T1

.

Quiz

```
type T1 is range 1 .. 100;
procedure Proc_A (X : in out T1);

type T2 is new T1 range 2 .. 99;
procedure Proc_B (X : in out T1);
procedure Proc_B (X : in out T2);

-- Other scope
procedure Proc_C (X : in out T2);

type T3 is new T2 range 3 .. 98;

procedure Proc_C (X : in out T3);
```

Which are T1's primitives

A. *Proc_A*
B. Proc_B
C. Proc_C
D. No primitives of T1

Explanations

A. Correct
B. Freeze: T1 has been derived
C. Freeze: scope change
D. Incorrect

.

Summary

Summary

- *Primitive* of a type
  - Subprogram above **freeze-point** that takes or return the type
  - Can be a primitive for **multiple types**
- Freeze point rules can be tricky
- Simple type derivation
  - Types derived from other types can only **add limitations**
    - Constraints, ranges
    - Cannot change underlying structure

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

- Four `for` variants
  - Index-based `in` or component-based `of`
  - Existential `some` or universal `all`
- Using arrow => to indicate *predicate* expression

```
(for some Index in Subtype_T => Predicate (Index))
(for all Index in Subtype_T => Predicate (Index))
(for some Value of Container_Obj => Predicate (Value))
(for all Value of Container_Obj => Predicate (Value))
```

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 /= ' ');*

```
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

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

Which piece(s) of code correctly perform(s) 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 => E1 =
E2));
```

D.
```
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 correctly perform(s) 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 => E1 = E2));`

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

B. Counterexample: `A = B = (0, 1, 0)` returns 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
  - This is the same lab as the *Expressions* lab, we're just replacing the validation functions with quantified expressions!
    - So you can just copy that project and update the code!

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

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

Creating Values

- Initialization is not assignment (but looks like it)!
- Via **limited constructor functions**
  - Functions returning values of limited types
- Via an **aggregate**
  - *limited aggregate* when used for a `limited` type

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

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(are) 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(are) 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

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

Ada 2005

```
--  Implicitly 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;

function F return T is
begin
   -- F body...
end F;

O : T := F;
```

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

A. `return Return : T := (I => 1)`

B. `return Result : T`

C. `return Value := (others => 1)`

D.
```
return R : T do
   R.I := 1;
end return;
```

Quiz

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

function F return T is
begin
   -- F body...
end F;

O : T := F;
```

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

A. `return Return : T := (I => 1)`

B. ***`return Result : T`***

C. `return Value := (others => 1)`

D. ***Correct:***
```
return R : T do
   R.I := 1;
end return;
```

A. Using **`return`** reserved keyword
B. OK, default value
C. Extended return must specify type
D. OK

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)

```
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;
   -- Complete Here
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 Priv is record
  Field : Integer := Lim'Size;
end record;
```

Quiz

```
package P is
   type Priv is private;
private
   type Lim is limited null record;
   -- Complete Here
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 Priv is record
   Field : Integer := Lim'Size;
end record;
```

A. E has limited type, partial view of Priv must be **limited private**
B. F has limited type, partial view of Priv must be **limited private**

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 a unique value generated 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)

```
package Employee_Data is

   subtype Name_T is String (1 .. 6);
   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 : Name_T;
                    Rate : Hourly_Rate_T := 0.0)
                    return Employee_T;
   function Id (Employee : Employee_T)
               return Id_T;
   function Name (Employee : Employee_T)
                 return Name_T;
   function Rate (Employee : Employee_T)
                 return Hourly_Rate_T;

private
   type Employee_T is limited record
      Name : Name_T        := (others => ' ');
      Rate : Hourly_Rate_T := 0.0;
      Id   : Id_T          := Id_T'First;
   end record;
end Employee_Data;
```

Limited Types Lab Solution - Timecards (Spec)

```ada
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  : Employee_Data.Name_T;
                    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 Employee_Data.Name_T;
   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 : Name_T;
                    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 := 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 Name_T is
      (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  : Employee_Data.Name_T;
                    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 Employee_Data.Name_T 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  ",
         Rate  => 1.1,
         Hours => 2.2);
   Two : constant Timecards.Timecard_T := Timecards.Create
        (Name  => "Barney",
         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

Private Types

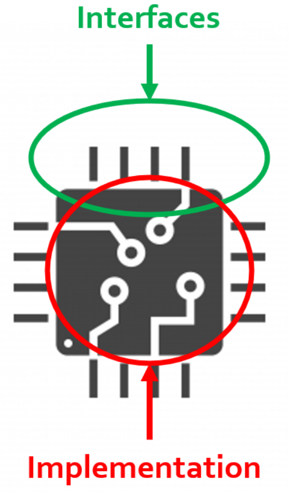
Introduction

Introduction

- Why does fixing bugs introduce new ones?
- Control over visibility is a primary factor
  - Changes to an abstraction's internals shouldn't break users
  - Including type representation
- Need tool-enforced rules to isolate dependencies
  - Between implementations of abstractions and their users
  - In other words, "information hiding"

Information Hiding

- A design technique in which implementation artifacts are made inaccessible to users
- Based on control of visibility to those artifacts
  - A product of "encapsulation"
  - Language support provides rigor
- Concept is "software integrated circuits"



Views

- Specify legal manipulation for objects of a type
  - Types are characterized by permitted values and operations
- Some views are implicit in language
  - Mode `in` parameters have a view disallowing assignment
- Views may be explicitly specified
  - Disallowing access to representation
  - Disallowing assignment
- Purpose: control usage in accordance with design
  - Adherence to interface
  - Abstract Data Types

Implementing Abstract Data Types via Views

Implementing Abstract Data Types

- A combination of constructs in Ada
- Not based on single "class" construct, for example
- Constituent parts
  - Packages, with "private part" of package spec
  - "Private types" declared in packages
  - Subprograms declared within those packages

Package Visible and Private Parts for Views

- Declarations in visible part are exported to users
- Declarations in private part are hidden from users
  - No compilable references to type's actual representation

```
package name is
... exported declarations of types, variables, subprograms ...
private
... hidden declarations of types, variables, subprograms ...
end name;
```

Declaring Private Types for Views

- Partial syntax

```
type defining_identifier is private;
```

- Private type declaration must occur in visible part
  - *Partial view*
  - Only partial information on the type
  - Users can reference the type name
    - But cannot create an object of that type until after the full type declaration
- Full type declaration must appear in private part
  - Completion is the *Full view*
  - **Never** visible to users
  - **Not** visible to designer until reached

```
package Control is
  type Valve is private;
  procedure Open (V : in out Valve);
  procedure Close (V : in out Valve);
  ...
private
  type Valve is ...
end Control;
```

Partial and Full Views of Types

- Private type declaration defines a *partial view*
  - The type name is visible
  - Only designer's operations and some predefined operations
  - No references to full type representation
- Full type declaration defines the *full view*
  - Fully defined as a record type, scalar, imported type, etc...
  - Just an ordinary type within the package
- Operations available depend upon one's view

Software Engineering Principles

- Encapsulation and abstraction enforced by views
  - Compiler enforces view effects
- Same protection as hiding in a package body
  - Recall "Abstract Data Machines" idiom
- Additional flexibility of types
  - Unlimited number of objects possible
  - Passed as parameters
  - Components of array and record types
  - Dynamically allocated
  - et cetera

Users Declare Objects of the Type

- Unlike "abstract data machine" approach
- Hence must specify which stack to manipulate
  - Via parameter

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

Compile-Time Visibility Protection

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

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

Benefits of Views

- Users depend only on visible part of specification
  - Impossible for users to compile references to private part
  - Physically seeing private part in source code is irrelevant
- Changes to implementation don't affect users
  - No editing changes necessary for user code
- Implementers can create bullet-proof abstractions
  - If a facility isn't working, you know where to look
- Fixing bugs is less likely to introduce new ones

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component is legal?

A. `Field_A : Integer := Private_T'Pos (Private_T'First);`

B. `Field_B : Private_T := null;`

C. `Field_C : Private_T := 0;`

D. `Field_D : Integer := Private_T'Size;`

```
    end record;
```

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component is legal?

A. `Field_A : Integer := Private_T'Pos (Private_T'First);`

B. `Field_B : Private_T := null;`

C. `Field_C : Private_T := 0;`

D. *`Field_D : Integer := Private_T'Size;`*

```
    end record;
```

Explanations

A. Visible part does not know `Private_T` is discrete
B. Visible part does not know possible values for `Private_T`
C. Visible part does not know possible values for `Private_T`
D. Correct - type will have a known size at run-time

Private Part Construction

Private Part Location

- Must be in package specification, not body
- Body usually compiled separately after declaration
- Users can compile their code before the package body is compiled or even written
  - Package definition

```
package Bounded_Stacks is
  type Stack is private;
  ...
private
  type Stack is ...
end Bounded_Stacks;
```

  - Package reference

```
with Bounded_Stacks;
procedure User is
  S : Bounded_Stacks.Stack;
...
begin
  ...
end User;
```

Private Part and Recompilation

- Private part is part of the specification
  - Compiler needs info from private part for users' code, e.g., storage layouts for private-typed objects
- Thus changes to private part require user recompilation
- Some vendors avoid "unnecessary" recompilation
  - Comment additions or changes
  - Additions which nobody yet references

Declarative Regions

- Declarative region of the spec extends to the body
  - Anything declared there is visible from that point down
  - Thus anything declared in specification is visible in body

```
package Foo is
   type Private_T is private;
   procedure X (B : in out Private_T);
private
   -- Y and Hidden_T are not visible to users
   procedure Y (B : in out Private_T);
   type Hidden_T is ...;
   type Private_T is array (1 .. 3) of Hidden_T;
end Foo;

package body Foo is
   -- Z is not visible to users
   procedure Z (B : in out Private_T) is ...
   procedure Y (B : in out Private_T) is ...
   procedure X (B : in out Private_T) is ...
 end Foo;
```

Full Type Declaration

- May be any type
  - Predefined or user-defined
  - Including references to imported types
- Contents of private part are unrestricted
  - Anything a package specification may contain
  - Types, subprograms, variables, etc.

```
package P is
  type T is private;
  ...
private
  type Vector is array (1.. 10)
     of Integer;
  function Initial
     return Vector;
  type T is record
    A, B : Vector := Initial;
  end record;
end P;
```

Deferred Constants

- Visible constants of a hidden representation
  - Value is "deferred" to private part
  - Value must be provided in private part
- Not just for private types, but usually so

```
package P is
  type Set is private;
  Null_Set : constant Set; -- exported name
  ...
private
  type Index is range ...
  type Set is array (Index) of Boolean;
  Null_Set : constant Set :=  -- definition
     (others => False);
end P;
```

Quiz

```
package P is
   type Private_T is private;
   Object_A : Private_T;
   procedure Proc (Param : in out Private_T);
private
   type Private_T is new Integer;
   Object_B : Private_T;
end package P;

package body P is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition is **not** legal?

A. `Object_A`
B. `Object_B`
C. `Object_C`
D. None of the above

Quiz

```
package P is
   type Private_T is private;
   Object_A : Private_T;
   procedure Proc (Param : in out Private_T);
private
   type Private_T is new Integer;
   Object_B : Private_T;
end package P;

package body P is
   Object_C : Private_T;
   procedure Proc (Param : in out Private_T) is null;
end P;
```

Which object definition is **not** legal?

A. *`Object_A`*
B. `Object_B`
C. `Object_C`
D. None of the above

An object cannot be declared until its type is fully declared. `Object_A` could be declared constant, but then it would have to be finalized in the `private` section.

View Operations

View Operations

- A matter of inside versus outside the package
  - Inside the package the view is that of the designer
  - Outside the package the view is that of the user

- **User** of package has **Partial** view
  - Operations exported by package
  - Basic operations

- **Designer** of package has **Full** view
  - **Once** completion is reached
  - All operations based upon full definition of type
  - Indexed components for arrays
  - components for records
  - Type-specific attributes
  - Numeric manipulation for numerics
  - et cetera

Designer View Sees Full Declaration

```
package Bounded_Stacks is
  Capacity : constant := 100;
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  ...
private
  type Index is range 0 .. Capacity;
  type Vector is array (Index range 1..Capacity) of Integer;
  type Stack is record
     Top : Integer;
     ...
end Bounded_Stacks;
```

Designer View Allows All Operations

```
package body Bounded_Stacks is
  procedure Push (Item : in Integer;
                  Onto : in out Stack) is
  begin
     Onto.Top := Onto.Top + 1;
     ...
  end Push;

  procedure Pop (Item : out Integer;
                 From : in out Stack) is
  begin
     Onto.Top := Onto.Top - 1;
     ...
  end Pop;
end Bounded_Stacks;
```

Users Have the Partial View

- Since they are outside package
- Basic operations
- Exported subprograms

```
package Bounded_Stacks is
  type Stack is private;
  procedure Push (Item : in Integer; Onto : in out Stack);
  procedure Pop (Item : out Integer; From : in out Stack);
  function Empty (S : Stack) return Boolean;
  procedure Clear (S : in out Stack);
  function Top (S : Stack) return Integer;
private
   ...
end Bounded_Stacks;
```

User View's Activities

- Declarations of objects
  - Constants and variables
  - Must call designer's functions for values

```
C : Complex.Number := Complex.I;
```

- Assignment, equality and inequality, conversions
- Designer's declared subprograms
- User-declared subprograms
  - Using parameters of the exported private type
  - Dependent on designer's operations

User View Formal Parameters

- Dependent on designer's operations for manipulation
  - Cannot reference type's representation
- Can have default expressions of private types

```
-- external implementation of "Top"
procedure Get_Top (
    The_Stack : in out Bounded_Stacks.Stack;
    Value : out Integer) is
  Local : Integer;
begin
  Bounded_Stacks.Pop (Local, The_Stack);
  Value := Local;
  Bounded_Stacks.Push (Local, The_Stack);
end Get_Top;
```

Limited Private

- **limited** is itself a view
  - Cannot perform assignment, copy, or equality
- **limited private** can restrain user's operation
  - Actual type **does not** need to be **limited**

```
package UART is
    type Instance is limited private;
    function Get_Next_Available return Instance;
[...]

declare
   A, B := UART.Get_Next_Available;
begin
   if A = B -- Illegal
   then
       A := B; -- Illegal
   end if;
```

When To Use or Avoid Private Types

When To Use Private Types

- Implementation may change
  - Allows users to be unaffected by changes in representation
- Normally available operations do not "make sense"
  - Normally available based upon type's representation
  - Determined by intent of ADT

```
A : Valve;
B : Valve;
C : Valve;
...
C := A + B;  -- addition not meaningful
```

- Users have no "need to know"
  - Based upon expected usage

When To Avoid Private Types

- If the abstraction is too simple to justify the effort
  - But that's the thinking that led to Y2K rework
- If normal user interface requires representation-specific operations that cannot be provided
  - Those that cannot be redefined by programmers
  - Would otherwise be hidden by a private type
  - If **Vector** is private, indexing of elements is annoying

```
type Vector is array (Positive range <>) of Float;
V : Vector (1 .. 3);
...
V (1) := Alpha;
```

Idioms

Effects of Hiding Type Representation

- Makes users independent of representation
  - Changes cannot require users to alter their code
  - Software engineering is all about money...
- Makes users dependent upon exported operations
  - Because operations requiring representation info are not available to users
    - Expression of values (aggregates, etc.)
    - Assignment for limited types
- Common idioms are a result
  - *Constructor*
  - *Selector*

Constructors

- Create designer's objects from user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Make (Real_Part : Float; Imaginary : Float) return Number
private
  type Number is record ...
end Complex;

package body Complex is
   function Make (Real_Part : Float; Imaginary_Part : Float)
     return Number is ...
end Complex;
...
A : Complex.Number :=
    Complex.Make (Real_Part => 2.5, Imaginary => 1.0);
```

Procedures As Constructors

- Spec

```
package Complex is
  type Number is private;
  procedure Make (This : out Number;  Real_Part, Imaginary : in Float) ;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;
```

- Body (partial)

```
package body Complex is
  procedure Make (This : out Number;
                  Real_Part, Imaginary : in Float) is
    begin
      This.Real_Part := Real_Part;
      This.Imaginary := Imaginary;
    end Make;
...
```

Selectors

- Decompose designer's objects into user's values
- Usually functions

```
package Complex is
  type Number is private;
  function Real_Part (This: Number) return Float;
  ...
private
  type Number is record
    Real_Part, Imaginary : Float;
  end record;
end Complex;

package body Complex is
  function Real_Part (This : Number) return Float is
  begin
    return This.Real_Part;
  end Real_Part;
  ...
end Complex;
...
Phase : Complex.Number := Complex.Make (10.0, 5.5);
Object : Float := Complex.Real_Part (Phase);
```

Lab

Private Types Lab

- Requirements
  - Implement a program to create a map such that
    - Map key is a description of a flag
    - Map element content is the set of colors in the flag
  - Operations on the map should include: Add, Remove, Modify, Get, Exists, Image
  - Main program should print out the entire map before exiting
- Hints
  - Should implement a **map** ADT (to keep track of the flags)
    - This **map** will contain all the flags and their color descriptions
  - Should implement a **set** ADT (to keep track of the colors)
    - This **set** will be the description of the map element
  - Each ADT should be its own package
  - At a minimum, the **map** and **set** type should be **private**

Private Types Lab Solution - Color Set

```
package Colors is
   type Color_T is (Red, Yellow, Green, Blue, Black);
   type Color_Set_T is private;

   Empty_Set : constant Color_Set_T;

   procedure Add (Set   : in out Color_Set_T;
                  Color :        Color_T);
   procedure Remove (Set   : in out Color_Set_T;
                     Color :        Color_T);
   function Image (Set : Color_Set_T) return String;
private
   type Color_Set_Array_T is array (Color_T) of Boolean;
   type Color_Set_T is record
      Values : Color_Set_Array_T := (others => False);
   end record;
   Empty_Set : constant Color_Set_T := (Values => (others => False));
end Colors;

package body Colors is
   procedure Add (Set   : in out Color_Set_T;
                  Color :        Color_T) is
   begin
      Set.Values (Color) := True;
   end Add;
   procedure Remove (Set   : in out Color_Set_T;
                     Color :        Color_T) is
   begin
      Set.Values (Color) := False;
   end Remove;

   function Image (Set   : Color_Set_T;
                   First : Color_T;
                   Last  : Color_T)
                   return String is
      Str : constant String := (if Set.Values (First) then Color_T'Image (First) else "");
   begin
      if First = Last then
         return Str;
      else
         return Str & " " & Image (Set, Color_T'Succ (First), Last);
      end if;
   end Image;
   function Image (Set : Color_Set_T) return String is
      (Image (Set, Color_T'First, Color_T'Last));
end Colors;
```

Private Types Lab Solution - Flag Map (Spec)

```
with Colors;
package Flags is
   type Key_T is (USA, England, France, Italy);
   type Map_Element_T is private;
   type Map_T is private;

   procedure Add (Map         : in out Map_T;
                  Key         :        Key_T;
                  Description :        Colors.Color_Set_T;
                  Success     :    out Boolean);
   procedure Remove (Map     : in out Map_T;
                     Key     :        Key_T;
                     Success :    out Boolean);
   procedure Modify (Map         : in out Map_T;
                     Key         :        Key_T;
                     Description :        Colors.Color_Set_T;
                     Success     :    out Boolean);

   function Exists (Map : Map_T; Key : Key_T) return Boolean;
   function Get (Map : Map_T; Key : Key_T) return Map_Element_T;
   function Image (Item : Map_Element_T) return String;
   function Image (Flag : Map_T) return String;
private
   type Map_Element_T is record
      Key         : Key_T := Key_T'First;
      Description : Colors.Color_Set_T := Colors.Empty_Set;
   end record;
   type Map_Array_T is array (1 .. 100) of Map_Element_T;
   type Map_T is record
      Values : Map_Array_T;
      Length : Natural := 0;
   end record;
end Flags;
```

Private Types Lab Solution - Flag Map (Body - 1 of 2)

```
procedure Add (Map         : in out Map_T;
               Key         :        Key_T;
               Description :        Colors.Color_Set_T;
               Success     :    out Boolean) is
begin
   Success := (for all Item of Map.Values
        (1 .. Map.Length) => Item.Key /= Key);
   if Success then
      declare
         New_Item : constant Map_Element_T :=
           (Key => Key, Description => Description);
      begin
         Map.Length              := Map.Length + 1;
         Map.Values (Map.Length) := New_Item;
      end;
   end if;
end Add;
procedure Remove (Map     : in out Map_T;
                  Key     :        Key_T;
                  Success :    out Boolean) is
begin
   Success := False;
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         Map.Values
           (I .. Map.Length - 1) := Map.Values
             (I + 1 .. Map.Length);
         Map.Length := Map.Length - 1;
         Success := True;
         exit;
      end if;
   end loop;
end Remove;
```

Private Types Lab Solution - Flag Map (Body - 2 of 2)

```
procedure Modify (Map         : in out Map_T;
                  Key         :        Key_T;
                  Description :        Colors.Color_Set_T;
                  Success     :    out Boolean) is
begin
   Success := False;
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         Map.Values (I).Description := Description;
         Success                    := True;
         exit;
      end if;
   end loop;
end Modify;
function Exists (Map : Map_T; Key : Key_T) return Boolean is
   (for some Item of Map.Values (1 .. Map.Length) => Item.Key = Key);
function Get (Map : Map_T; Key : Key_T) return Map_Element_T is
   Ret_Val : Map_Element_T;
begin
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         Ret_Val := Map.Values (I);
         exit;
      end if;
   end loop;
   return Ret_Val;
end Get;
function Image (Item : Map_Element_T) return String is
  (Key_T'Image (Item.Key) & " => " & Colors.Image (Item.Description));
function Image (Flag : Map_T) return String is
   Ret_Val : String (1 .. 1_000);
   Next    : Integer := Ret_Val'First;
begin
   for Item of Flag.Values (1 .. Flag.Length) loop
      declare
         Str : constant String := Image (Item);
      begin
         Ret_Val (Next .. Next + Str'Length) := Image (Item) & ASCII.LF;
         Next                           := Next + Str'Length + 1;
      end;
   end loop;
   return Ret_Val (1 .. Next - 1);
end Image;
```

Private Types Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Colors;
with Flags;
with Input;
procedure Main is
   Map : Flags.Map_T;
begin

   loop
      Put ("Enter country name (");
      for Key in Flags.Key_T loop
         Put (Flags.Key_T'Image (Key) & " ");
      end loop;
      Put ("): ");
      declare
         Str         : constant String := Get_Line;
         Key         : Flags.Key_T;
         Description : Colors.Color_Set_T;
         Success     : Boolean;
      begin
         exit when Str'Length = 0;
         Key         := Flags.Key_T'Value (Str);
         Description := Input.Get;
         if Flags.Exists (Map, Key) then
            Flags.Modify (Map, Key, Description, Success);
         else
            Flags.Add (Map, Key, Description, Success);
         end if;
      end;
   end loop;

   Put_Line (Flags.Image (Map));
end Main;
```

Summary

Summary

- Tool-enforced support for Abstract Data Types
  - Same protection as Abstract Data Machine idiom
  - Capabilities and flexibility of types
- May also be `limited`
  - Thus additionally no assignment or predefined equality
  - More on this later
- Common interface design idioms have arisen
  - Resulting from representation independence
- Assume private types as initial design choice
  - Change is inevitable

Access Types

Introduction

Access Types Design

- Memory-addressed objects are called *access types*
- Objects are associated to *pools* of memory
  - With different allocation / deallocation policies
- Access objects are **guaranteed** to always be meaningful
  - In the absence of `Unchecked_Deallocation`
  - And if pool-specific

- Ada

```
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_A : Integer_General_Access := G'access;
```

- C++

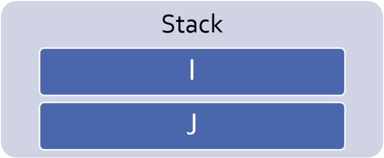
```
int * P_C = malloc (sizeof (int));
int * P_CPP = new int;
int * G_C = &Some_Int;
```

Access Types Can Be Dangerous

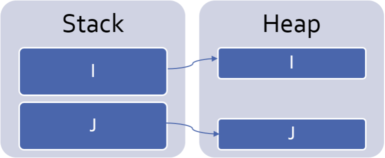
- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot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

Stack vs Heap

```
I : Integer := 0;
J : String := "Some Long String";
```




```
I : Access_Int := new Integer'(0);
J : Access_Str := new String'("Some Long String");
```



Access Types

Declaration Location

- Can be at library level

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

- Can be nested in a procedure

```
package body P is
   procedure Proc is
      type String_Access is access 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)

Null Values

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

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

Access Types and Primitives

- Subprogram 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
```

Dereferencing Access Types

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

Dereference Examples

```
type R is record
  F1, F2 : Integer;
end record;
type A_Int is access Integer;
type A_String is access all String;
type A_R is access R;
V_Int    : A_Int := new Integer;
V_String : A_String := new String'("abc");
V_R      : A_R := new R;

V_Int.all := 0;
V_String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V_R.all := (0, 0);
V_R.F1 := 1; -- similar to V_R.all.F1 := 1;
```

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 access
- But sometimes, you have to do what you have to do ...
  - There's no simple way of doing it
  - Ada provides **Ada.Unchecked_Deallocation**
  - Has to be instantiated (it's a generic)
  - Must work on an object, reset to `null` afterwards

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**

Aliased Objects Examples

```
type Acc is access all Integer;
V, G : Acc;
I : aliased Integer;
...
V := I'Access;
V.all := 5; -- Same a I := 5
...
procedure P1 is
   I : aliased Integer;
begin
   G := I'Unchecked_Access;
   P2;
end P1;

procedure P2 is
begin
   -- OK when P2 called from P1.
   -- What if P2 is called from elsewhere?
   G.all := 5;
end P2;
```

Quiz

```
type One_T is access all Integer;
type Two_T is access Integer;

A : aliased Integer;
B : Integer;

One : One_T;
Two : Two_T;
```

Which assignment is legal?

A. `One := B'Access;`
B. `One := A'Access;`
C. `Two := B'Access;`
D. `Two := A'Access;`

Quiz

```
type One_T is access all Integer;
type Two_T is access Integer;

A : aliased Integer;
B : Integer;

One : One_T;
Two : Two_T;
```

Which assignment is legal?

A. `One := B'Access;`
B. *`One := A'Access;`*
C. `Two := B'Access;`
D. `Two := A'Access;`

`'Access` is only allowed for general access types (`One_T`). To use `'Access` on an object, the object must be **`aliased`**.

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
   O0 : aliased Integer;
   procedure Proc is
      --  Library level subprogram, depth 1
      type Acc1 is access all Integer;
      procedure Nested is
         -- Nested subprogram, enclosing + 1, here 2
         O2 : aliased Integer;
```

- Objects can be referenced by access **types** that are at **same depth or deeper**
  - An **access scope** must be $\leq$ the object scope
- `type Acc1` (depth 1) can access `O0` (depth 0) but not **O2** (depth 2)
- The compiler checks it statically
  - Removing checks is a workaround!
- Note: Subprogram library units are at **depth 1** and not 0

Introduction to Accessibility Checks (2/2)

```
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;
```

- To avoid having to face these issues, avoid nested access types

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);
   G := null; -- This is "reasonable"
end P;
```

Using Access Types 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;
```

Quiz

```
type Global_Access_T is access all Integer;
Global_Pointer : Global_Access_T;
Global_Object  : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Pointer : Local_Access_T;
   Local_Object  : aliased Integer;
begin
```

Which assignment is **not** legal?

A. `Global_Pointer := Global_Object'Access;`
B. `Global_Pointer := Local_Object'Access;`
C. `Local_Pointer  := Global_Object'Access;`
D. `Local_Pointer  := Local_Object'Access;`

Quiz

```
type Global_Access_T is access all Integer;
Global_Pointer : Global_Access_T;
Global_Object  : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Pointer : Local_Access_T;
   Local_Object  : aliased Integer;
begin
```

Which assignment is **not** legal?

A. `Global_Pointer := Global_Object'Access;`
B. *`Global_Pointer := Local_Object'Access;`*
C. `Local_Pointer  := Global_Object'Access;`
D. `Local_Pointer  := Local_Object'Access;`

Explanations

A. Pointer type has same depth as object
B. Pointer type is not allowed to have higher level than pointed-to object
C. Pointer type has lower depth than pointed-to object
D. Pointer type has same depth as object

Memory Management

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...

Anonymous Access Types

Anonymous Access Parameters

- Parameter modes are of 4 types: **in**, **out**, **in out**, **access**
- The access mode is called *anonymous access type*
  - Anonymous access is implicitly general (no need for **all**)
- When used:
  - Any named access can be passed as parameter
  - Any anonymous access can be passed as parameter

```
type Acc is access all Integer;
Aliased_Integer : aliased Integer;
Access_Object   : Acc := Aliased_Integer'access;
procedure P1 (Anon_Access : access Integer) is null;
procedure P2 (Access_Parameter : access Integer) is
begin
   P1 (Aliased_Integer'access);
   P1 (Access_Object);
   P1 (Access_Parameter);
end P2;
```

Anonymous Access Types

- Other places can declare an anonymous access

```
function F return access Integer;
V : access Integer;
type T (V : access Integer) is record
  C : access Integer;
end record;
type A is array (Integer range <>) of access Integer;
```

- Do not use them without a clear understanding of accessibility check rules

Anonymous Access Constants

- **constant** (instead of **all**) denotes an access type through which the referenced object cannot be modified

```
type CAcc is access constant Integer;
G1 : aliased Integer;
G2 : aliased constant Integer := 123;
V1 : CAcc := G1'Access;
V2 : CAcc := G2'Access;
V1.all := 0; -- illegal
```

- **not null** denotes an access type for which null value cannot be accepted
  - Available in Ada 2005 and later

```
type NAcc is not null access Integer;
V : NAcc := null; -- illegal
```

- Also works for subprogram parameters

```
procedure Bar (V1 : access constant Integer);
procedure Foo (V1 : not null access Integer); -- Ada 2005
```

Lab

Access Types Lab

- Overview
  - Create a (really simple) Password Manager
    - The Password Manager should store the password and a counter for each of some number of logins
    - As it's a Password Manager, you want to modify the data directly (not pass the information around)
- Requirements
  - Create a Password Manager package
    - Create a record to store the password string and the counter
    - Create an array of these records indexed by the login identifier
    - The user should be able to retrieve a pointer to the record, either for modification or for viewing
  - Main program should:
    - Set passwords and initial counter values for many logins
    - Print password and counter value for each login
- Hint
  - Password is a string of varying length
    - Easiest way to do this is a pointer to a string that gets initialized to the correct length

Access Types Lab Solution - Password Manager

```
package Password_Manager is

   type Login_T is (Email, Banking, Amazon, Streaming);
   type Password_T is record
      Count    : Natural;
      Password : access String;
   end record;

   type Modifiable_T is access all Password_T;
   type Viewable_T is access constant Password_T;

   function Update (Login : Login_T) return Modifiable_T;
   function View (Login : Login_T) return Viewable_T;

end Password_Manager;

package body Password_Manager is

   Passwords : array (Login_T) of aliased Password_T;

   function Update (Login : Login_T) return Modifiable_T is
      (Passwords (Login)'Access);
   function View (Login : Login_T) return Viewable_T is
      (Passwords (Login)'Access);

end Password_Manager;
```

Access Types Lab Solution - Main

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

   procedure Update (Which : Password_Manager.Login_T;
                     Pw    : String;
                     Count : Natural) is
   begin
      Update (Which).Password := new String'(Pw);
      Update (Which).Count    := Count;
   end Update;

begin
   Update (Email, "QWE!@#", 1);
   Update (Banking, "asd123", 22);
   Update (Amazon, "098poi", 333);
   Update (Streaming, ")(*LKJ", 444);

   for Login in Login_T'Range loop
      Put_Line
        (Login'Image & " => " & View (Login).Password.all &
         View (Login).Count'Image);
   end loop;
end Main;
```

Summary

Summary

- Access types are the same as C/C++ pointers
- There are usually better ways of memory management
  - Language has its own ways of dealing with large objects passed as parameters
  - Language has libraries dedicated to memory allocation / deallocation
- At a minimum, create your own generics to do allocation / deallocation
  - Minimize memory leakage and corruption

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

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

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

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

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 (S : Stack_T);
```

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 (S, 1);
Integer_Stack_Utils.Print (S);
```

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;
   type T2 (<>) is private;
   type T3 is limited private;
package Parent is
```

- The actual parameter must be no more restrictive then the *generic contract*

Generic Types Parameters (2/3)

- Generic formal parameter tells generic what it is allowed to do with the type

`type T1 is (<>);`	Discrete type; `'First`, `'Succ`, etc available
`type T2 is range <>;`	Signed Integer type; appropriate mathematic operations allowed
`type T3 is digits <>;`	Floating point type; appropriate mathematic operations allowed
`type T4 (<>);`	Indefinite type; can only be used as target of `access`
`type T5 is tagged;`	`tagged` type; can extend the type
`type T6 is private;`	No knowledge about the type other than assignment, comparison, object creation allowed
`type T7 (<>) is private;`	`(<>)` indicates type can be unconstrained, so any object has to be initialized

Generic Types Parameters (3/3)

- The usage in the generic has to follow the contract
  - Generic Subprogram

```
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
```

  - Instantiations

```
type Limited_T is limited null record;

-- unconstrained types are accepted
procedure P1 is new P (String);

-- type is already constrained
-- (but generic will still always initialize objects)
procedure P2 is new P (Integer);

-- Illegal: the type can't be limited because the generic
-- thinks it can make copies
procedure P3 is new P (Limited_T);
```

Generic Parameters Can Be Combined

- Consistency is checked at compile-time

```
generic
   type T (<>) is private;
   type Acc is access all T;
   type Index is (<>);
   type Arr is array (Index range <>) of Acc;
function Element (Source   : Arr;
                  Position : Index)
                  return T;

type String_Ptr is access all String;
type String_Array is array (Integer range <>)
    of String_Ptr;

function String_Element is new Element
   (T     => String,
    Acc   => String_Ptr,
    Index => Integer,
    Arr   => String_Array);
```

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is **not** a legal 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 **not** a legal 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

Generic Constants/Variables as 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 package

```
generic
  type Element_T is private;
  Array_Size     : Positive;
  High_Watermark : in out Element_T;
package Repository is
```

- Generic instance

```
V   : Float;
Max : Float;

procedure My_Repository is new Repository
  (Element_T      => Float,
   Array_size     => 10,
   High_Watermark => Max);
```

Generic Subprogram Parameters

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

```
generic
   type T is private;
   with function Less_Than (L, R : T) return Boolean;
function Max (L, R : T) return T;

function Max (L, R : T) return T is
begin
   if Less_Than (L, R) then
      return R;
   else
      return L;
   end if;
end Max;

type Something_T is null record;
function Less_Than (L, R : Something_T) return Boolean;
procedure My_Max is new Max (Something_T, Less_Than);
```

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
  type T is private;
  with function Is_Valid (P : T) return Boolean is <>;
  with procedure Error_Message (P : T) is null;
procedure Validate (P : T);

function Is_Valid_Record (P : Record_T) return Boolean;

procedure My_Validate is new Validate (Record_T,
                                       Is_Valid_Record);
-- Is_Valid maps to Is_Valid_Record
-- Error_Message maps to a null subprogram
```

Quiz

```
generic
   type Element_T is (<>);
   Last : in out Element_T;
procedure Write (P : Element_T);

Numeric        : Integer;
Enumerated     : Boolean;
Floating_Point : Float;
```

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

A. `procedure Write_A is new Write (Integer, Numeric)`
B. `procedure Write_B is new Write (Boolean, Enumerated)`
C. `procedure Write_C is new Write (Integer, Integer'Pos (Enumerated))`
D. `procedure Write_D is new Write (Float, Floating_Point)`

Quiz

```
generic
   type Element_T is (<>);
   Last : in out Element_T;
procedure Write (P : Element_T);

Numeric        : Integer;
Enumerated     : Boolean;
Floating_Point : Float;
```

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

A. *`procedure Write_A is new Write (Integer, Numeric)`*
B. *`procedure Write_B is new Write (Boolean, Enumerated)`*
C. `procedure Write_C is new Write (Integer, Integer'Pos (Enumerated))`
D. `procedure Write_D is new Write (Float, Floating_Point)`

A. Legal
B. Legal
C. The second generic parameter has to be a variable
D. The first generic parameter has to be discrete

Quiz

Ada 2005

```
procedure Double (X : in out Integer);
procedure Square (X : in out Integer);
procedure Half (X : in out Integer);
generic
   with procedure Double (X : in out Integer) is <>;
   with procedure Square (X : in out Integer) is null;
procedure Math (P : in out Integer);
procedure Math (P : in out Integer) is
begin
   Double(P);
   Square(P);
end Math;
procedure Instance is new Math (Double => Half);
Number : Integer := 10;
```

What is the value of Number after calling `Instance (Number)`

A. 20
B. 400
C. 5
D. 10

Quiz

Ada 2005

```
procedure Double (X : in out Integer);
procedure Square (X : in out Integer);
procedure Half (X : in out Integer);
generic
   with procedure Double (X : in out Integer) is <>;
   with procedure Square (X : in out Integer) is null;
procedure Math (P : in out Integer);
procedure Math (P : in out Integer) is
begin
   Double(P);
   Square(P);
end Math;
procedure Instance is new Math (Double => Half);
Number : Integer := 10;
```

What is the value of Number after calling `Instance (Number)`

A. 20
B. 400
C. ***5***
D. 10

A. Would be correct for `procedure Instance is new Math;`
B. Would be correct for either
   `procedure Instance is new Math (Double, Square);` *or*
   `procedure Instance is new Math (Square => Square);`
C. Correct
   - We call formal parameter `Double`, which has been assigned to actual subprogram `Half`, so `P`, which is 10, is halved.
   - Then we call formal parameter `Square`, which has no actual subprogram, so it defaults to `null`, so nothing happens to `P`
D. Would be correct for either
   `procedure Instance is new Math (Double, Half);` *or*
   `procedure Instance is new Math (Square => Half);`

Quiz Answer In Depth

A. Wrong - result for `procedure Instance is new Math;`
B. Wrong - result for
   `procedure Instance is new Math (Double, Square);`
C. `Double` at line 10 is mapped to `Half` at line 3, and `Square` at line 11 wasn't specified so it defaults to `null`
D. Wrong - result for
   `procedure Instance is new Math (Square => Half);`

Quiz Answer In Depth

A. Wrong - result for `procedure Instance is new Math;`
B. Wrong - result for
   `procedure Instance is new Math (Double, Square);`
C. `Double` at line 10 is mapped to `Half` at line 3, and `Square` at line 11 wasn't specified so it defaults to `null`
D. Wrong - result for
   `procedure Instance is new Math (Square => Half);`

`Math` is going to call two subprograms in order, `Double` and `Square`, but both of those come from the formal data.

Whatever is used for `Double`, will be called by the `Math` instance. If nothing is passed in, the compiler tries to find a subprogram named `Double` and use that. If it doesn't, that's a compile error.

Whatever is used for `Square`, will be called by the `Math` instance. If nothing is passed in, the compiler will treat this as a null call.

In our case, `Half` is passed in for the first subprogram, but nothing is passed in for the second, so that call will just be null.

So the final answer should be 5 (hence letter C).

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) legal for `G_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) legal for `G_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 record structure containing multiple fields
    - Need subprograms to convert the record to a string, and compare the order of two records
    - Lab prompt package `Data_Type` contains a framework
  - Create a generic list implementation
    - Need subprograms to add items to the list, sort the list, and print the list
  - The **main** program should:
    - Add many records to the list
    - Sort the list
    - Print the list
- Hints
  - Sort routine will need to know how to compare elements
  - Print routine will need to know how to print one element

Genericity Lab Solution - Generic (Spec)

```
generic
   type Element_T is private;
   Max_Size : Natural;
   with function ">" (L, R : Element_T) return Boolean is <>;
   with function Image (Element : Element_T) return String;
package Generic_List is

   type List_T is private;

   procedure Add (This : in out List_T;
                  Item : in     Element_T);
   procedure Sort (This : in out List_T);
   procedure Print (List : 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 record
      Values : List_Array_T;
      Length : Index_T := 0;
   end record;
end Generic_List;
```

Genericity Lab Solution - Generic (Body)

```
with Ada.Text_io; use Ada.Text_IO;
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 1 .. This.Length - I 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;

   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;
```

Genericity Lab Solution - Main

```
with Data_Type;
with Generic_List;
procedure Main is
   package List is new Generic_List (Element_T => Data_Type.Record_T,
                                     Max_Size  => 20,
                                     ">"       => Data_Type.">",
                                     Image => Data_Type.Image);

   My_List : List.List_T;
   Element : Data_Type.Record_T;

begin
   List.Add (My_List, (Integer_Field   => 111,
                       Character_Field => 'a'));
   List.Add (My_List, (Integer_Field    => 111,
                       Character_Field => 'z'));
   List.Add (My_List, (Integer_Field    => 111,
                       Character_Field => 'A'));
   List.Add (My_List, (Integer_Field    => 999,
                       Character_Field => 'B'));
   List.Add (My_List, (Integer_Field    => 999,
                       Character_Field => 'Y'));
   List.Add (My_List, (Integer_Field    => 999,
                       Character_Field => 'b'));
   List.Add (My_List, (Integer_Field    => 112,
                       Character_Field => 'a'));
   List.Add (My_List, (Integer_Field    => 998,
                       Character_Field => 'z'));

   List.Sort (My_List);
   List.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

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;
```

- Conversion 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
```

Overriding Indicators

Ada 2005

- Optional **overriding** and **not overriding** indicators

```
type Shape_T is tagged record
   Name : String(1..10);
end record;

-- primitives of "Shape_T"
function Get_Name (S : Shape_T) return String;
procedure Set_Name (S : in out Shape_T);

-- Derive "Point" from Shape_T
type Point is new Shape_T with record
   Origin : Coord_T;
end Point;

-- Get_Name is inherited
-- We want to _change_ the behavior of Set_Name
overriding procedure Set_Name (P : in out Point_T);
-- We want to _add_ a new primitive
not overriding procedure Set_Origin (P : in out Point_T);
```

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;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

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;
Object : T1;
procedure P (O : T1) is null;
```

D.
```
package Nested is
  type T1 is tagged null record;
end Nested;
use Nested;
procedure P (O : T1) is null;
```

A. Primitive (same scope)
B. Primitive (T1 is not yet frozen)
C. T1 is frozen by the object declaration
D. Primitive must be declared in same scope as type

Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

procedure Main is
   The_Shape : Shapes.Shape;
   The_Color : Colors.Color;
   The_Weight : Weights.Weight;
```

Which statement(s) is(are) valid?

A. `The_Shape.P`
B. `P (The_Shape)`
C. `P (The_Color)`
D. `P (The_Weight)`

Quiz

```
with Shapes; -- Defines tagged type Shape, with primitive P
with Colors; use Colors; -- Defines tagged type Color, with primitive P
with Weights; -- Defines tagged type Weight, with primitive P
use type Weights.Weight;

procedure Main is
   The_Shape : Shapes.Shape;
   The_Color : Colors.Color;
   The_Weight : Weights.Weight;
```

Which statement(s) is(are) valid?

A. *`The_Shape.P`*
B. `P (The_Shape)`
C. *`P (The_Color)`*
D. `P (The_Weight)`

D. `use type` only gives visibility to operators; needs to be `use all type`

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

Extending Tagged Types

How Do You Extend A Tagged Type?

- Premise of a tagged type is to *extend* an existing type
- In general, that means we want to add more fields
  - We can extend a **tagged** type by adding fields

```
package Animals is
  type Animal_T is tagged record
    Age : Natural;
  end record;
end Animals;

with Animals; use Animals;
package Mammals is
  type Mammal_T is new Animal_T with record
    Number_Of_Legs : Natural;
  end record;
end Mammals;

with Mammals; use Mammals;
package Canines is
  type Canine_T is new Mammal_T with record
    Domesticated : Boolean;
  end record;
end Canines;
```

Tagged Aggregate

- At initialization, all fields (including **inherited**) must have a **value**

```
Animal : Animal_T := (Age => 1);
Mammal : Mammal_T := (Age            => 2,
                      Number_Of_Legs => 2);
Canine : Canine_T := (Age            => 2,
                      Number_Of_Legs => 4,
                      Domesticated   => True);
```

- But we can also "seed" the aggregate with a parent object

```
Mammal := (Animal with Number_Of_Legs => 4);
Canine := (Animal with Number_Of_Legs => 4,
                       Domesticated   => False);
Canine := (Mammal with Domesticated => True);
```

Private Tagged Types

- But data hiding says types should be private!
- So we can define our base type as private

```
package Animals is
  type Animal_T is tagged private;
  function Get_Age (P : Animal_T) return Natural;
  procedure Set_Age (P : in out Animal_T; A : Natural);
private
  type Animal_T is tagged record
     Age : Natural;
  end record;
end Animals;
```

- And still allow derivation

```
with Animals;
package Mammals is
  type Mammal_T is new Animals.Animal_T with record
    Number_Of_Legs : Natural;
  end record;
```

- But now the only way to get access to `Age` is with accessor subprograms

Private Extensions

- In the previous slide, we exposed the fields for `Mammal_T`!
- Better would be to make the extension itself private

```
package Mammals is
  type Mammal_T is new Animals.Animal_T with private;
private
  type Mammal_T is new Animals.Animal_T with record
     Number_Of_Legs : Natural;
  end record;
end Mammals;
```

Aggregates with Private Tagged Types

- Remember, an aggregate must specify values for all components
  - But with private types, we can't see all the components!
- So we need to use the "seed" method:

```
procedure Inside_Mammals_Pkg is
  Animal : Animal_T := Animals.Create;
  Mammal : Mammal_T;
begin
  Mammal := (Animal with Number_Of_Legs => 4);
  Mammal := (Animals.Create with Number_Of_Legs => 4);
end Inside_Mammals_Pkg;
```

- Note that we cannot use `others => <>` for components that are not visible to us

```
Mammal := (Number_Of_Legs => 4,
           others         => <>);  -- Compile Error
```

Null Extensions

- To create a new type with no additional fields
  - We still need to "extend" the record - we just do it with an empty record

    ```
    type Dog_T is new Canine_T with null record;
    ```

- We still need to specify the "added" fields in an aggregate

```
C     : Canine_T := Canines.Create;
Dog1 : Dog_T := C; -- Compile Error
Dog2 : Dog_T := (C with null record);
```

Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of C is/are valid?

A. `function Create return Child_T is (Parents.Create with Count => 0);`
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. `function Create return Child_T is (P with Count => 0);`

Quiz

Given the following code:

```
package Parents is
  type Parent_T is tagged private;
  function Create return Parent_T;
private
  type Parent_T is tagged record
     Id : Integer;
  end record;
end Parents;

with Parents; use Parents;
package Children is
  P : Parent_T;
  type Child_T is new Parent_T with record
     Count : Natural;
  end record;
  function Create (C : Natural) return Child_T;
end Children;
```

Which completion(s) of C is/are valid?

A. ***`function Create return Child_T is (Parents.Create with Count => 0);`***
B. `function Create return Child_T is (others => <>);`
C. `function Create return Child_T is (0, 0);`
D. ***`function Create return Child_T is (P with Count => 0);`***

Explanations

A. Correct - `Parents.Create` returns `Parent_T`
B. Cannot use **`others`** to complete private part of an aggregate
C. Aggregate has no visibility to `Id` field, so cannot assign
D. Correct - `P` is a `Parent_T`

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 **(Ada 2005 and above)**
  - Data hiding is important!

Tagged Derivation Lab Solution - Types (Spec)

```
package Employee is
   type Person_T is tagged private;
   subtype Name_T is String (1 .. 6);
   type Date_T is record
      Year  : Positive;
      Month : Positive;
      Day   : Positive;
   end record;
   type Job_T is (Sales, Engineer, Bookkeeping);

   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T);
   function Name (O : Person_T) return Name_T;
   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T);
   function Birth_Date (O : Person_T) return Date_T;
   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 :        Date_T);
   not overriding function Start_Date (O : Employee_T) return Date_T;
   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 :        Job_T);
   not overriding function Job (O : Position_T) return Job_T;
   overriding procedure Print (O : Position_T);

private
   type Person_T is tagged record
      The_Name       : Name_T;
      The_Birth_Date : Date_T;
   end record;

   type Employee_T is new Person_T with record
      The_Employee_Id : Positive;
      The_Start_Date  : Date_T;
   end record;

   type Position_T is new Employee_T with record
      The_Job : Job_T;
   end record;
end Employee;
```

Tagged Derivation Lab Solution - Types (Partial Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Employee is

   function Image (Date : Date_T) return String is
     (Date.Year'Image & " -" & Date.Month'Image & " -" & Date.Day'Image);

   procedure Set_Name (O     : in out Person_T;
                       Value :        Name_T) is
   begin
      O.The_Name := Value;
   end Set_Name;
   function Name (O : Person_T) return Name_T is (O.The_Name);

   procedure Set_Birth_Date (O     : in out Person_T;
                             Value :        Date_T) is
   begin
      O.The_Birth_Date := Value;
   end Set_Birth_Date;
   function Birth_Date (O : Person_T) return Date_T is (O.The_Birth_Date);

   procedure Print (O : Person_T) is
   begin
      Put_Line ("Name: " & O.Name);
      Put_Line ("Birthdate: " & Image (O.Birth_Date));
   end Print;

   not overriding procedure Set_Start_Date (O     : in out Employee_T;
                                            Value :        Date_T) is
   begin
      O.The_Start_Date := Value;
   end Set_Start_Date;
   not overriding function Start_Date (O : Employee_T) return Date_T is
      (O.The_Start_Date);

   overriding procedure Print (O : Employee_T) is
   begin
      Print (Person_T (O)); --  Use parent "Print"
      Put_Line ("Startdate: " & Image (O.Start_Date));
   end Print;

```

Tagged Derivation Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Employee;
procedure Main is
   Applicant : Employee.Person_T;
   Employ    : Employee.Employee_T;
   Staff     : Employee.Position_T;

begin
   Applicant.Set_Name ("Wilma ");
   Applicant.Set_Birth_Date ((Year  => 1_234,
                              Month => 12,
                              Day   => 1));

   Employ.Set_Name ("Betty ");
   Employ.Set_Birth_Date ((Year  => 2_345,
                           Month => 11,
                           Day   => 2));
   Employ.Set_Start_Date ((Year  => 3_456,
                           Month => 10,
                           Day   => 3));

   Staff.Set_Name ("Bambam");
   Staff.Set_Birth_Date ((Year  => 4_567,
                          Month => 9,
                          Day   => 4));
   Staff.Set_Start_Date ((Year  => 5_678,
                          Month => 8,
                          Day   => 5));
   Staff.Set_Job (Employee.Engineer);

   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

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'Tag; -- False
B3 : Boolean := Child_Class'Tag = Parent'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

Warning: Subprograms with parameter of type **T'Class** are not primitives of **T**

```
type Root is tagged 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);
```

'Class and Prefix Notation

Ada 2012

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

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

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

Dispatching and Redispatching

Calls on class-wide types (1/3)

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

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

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

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

Calls on class-wide types (2/3)

- 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

```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++

```cpp
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 P; use P;
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 P; use P;
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

Multiple dispatching operands

- Primitives with multiple dispatching operands are allowed if all operands are of the same type

```
type Root is tagged null 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 tagged null 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;         -- Error - ambiguous expression
...
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 that has multiple shapes
    - Create a nested subprogram that takes any shape and prints all appropriate 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)

```
package Shapes is
   type Length_T is new Natural;
   type Lengths_T is array (Positive range <>) of Length_T;
   subtype Description_T is String (1 .. 10);

   type Shape_T is abstract tagged record
      Description : Description_T;
   end record;
   function Get_Description (Shape : Shape_T'Class) return Description_T;
   function Number_Of_Sides (Shape : Shape_T) return Natural is abstract;
   function Perimeter (Shape : Shape_T) return Length_T is abstract;

   type Quadrilateral_T is new Shape_T with record
      Lengths : Lengths_T (1 .. 4);
   end record;
   function Number_Of_Sides (Shape : Quadrilateral_T) return Natural;
   function Perimeter (Shape : Quadrilateral_T) return Length_T;

   type Square_T is new Quadrilateral_T with null record;
   function Perimeter (Shape : Square_T) return Length_T;

   type Triangle_T is new Shape_T with record
      Lengths : Lengths_T (1 .. 3);
   end record;
   function Number_Of_Sides (Shape : Triangle_T) return Natural;
   function Perimeter (Shape : Triangle_T) return Length_T;
end Shapes;
```

Polymorphism Lab Solution - Shapes (Body)

```
package body Shapes is

   function Perimeter (Lengths : Lengths_T) return Length_T is
      Ret_Val : Length_T := 0;
   begin
      for I in Lengths'First .. Lengths'Last
      loop
         Ret_Val := Ret_Val + Lengths (I);
      end loop;
      return Ret_Val;
   end Perimeter;

   function Get_Description (Shape : Shape_T'Class) return Description_T is
      (Shape.Description);

   function Number_Of_Sides (Shape : Quadrilateral_T) return Natural is
      (4);
   function Perimeter (Shape : Quadrilateral_T) return Length_T is
      (Perimeter (Shape.Lengths));

   function Perimeter (Shape : Square_T) return Length_T is
      (4 * Shape.Lengths (Shape.Lengths'First));

   function Number_Of_Sides (Shape : Triangle_T) return Natural is
      (3);
   function Perimeter (Shape : Triangle_T) return Length_T is
      (Perimeter (Shape.Lengths));
end Shapes;
```

Polymorphism Lab Solution - Main

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

   Rectangle : constant Shapes.Quadrilateral_T :=
     (Description => "rectangle ",
      Lengths     => (10, 20, 10, 20));
   Triangle : constant Shapes.Triangle_T :=
     (Description => "triangle  ",
      Lengths     => (200, 300, 400));
   Square : constant Shapes.Square_T :=
     (Description => "square    ",
      Lengths     => (5_000, 5_000, 5_000, 5_000));

   procedure Describe (Shape : Shapes.Shape_T'Class) is
   begin
      Put_Line (Shape.Get_Description);
      Put_Line
        ("  Number of sides:" & Integer'Image (Shape.Number_Of_Sides));
      Put_Line ("  Perimeter:" & Shapes.Length_T'Image (Shape.Perimeter));
   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;
   function Image (Coord : Coordinate_T) return String is
     ("(" & Coord.X_Coord'Image & "," &
            Coord.Y_Coord'Image & " )");

   type Line_T is array (1 .. 2) of Coordinate_T;
   type Lines_T is array (Natural range <>) of Line_T;

   type Color_T is mod 256;
   type Width_T is range 1 .. 10;

end Base_Types;
```

Inheritance Lab Solution - Shapes

```
with Base_Types;
package Geometry is

   -- Create a tagged type to define shapes
   type Object_T is abstract tagged private;

   -- Create accessor functions for some common component
   function Origin (Object : Object_T'Class) return Base_Types.Coordinate_T;

private

   type Object_T is abstract tagged record
      The_Origin : Base_Types.Coordinate_T;
   end record;

   function Origin (Object : Object_T'Class) return Base_Types.Coordinate_T is
      (Object.The_Origin);

end Geometry;
```

Inheritance Lab Solution - Drawing (Spec)

```
with Base_Types;
package Line_Draw is

   type Object_T is interface;

   -- Create accessor functions for some line attributes
   procedure Set_Color (Object : in out Object_T;
                        Color  : in     Base_Types.Color_T)
         is abstract;
   function Color (Object : Object_T) return Base_Types.Color_T
         is abstract;

   procedure Set_Pen_Width (Object : in out Object_T;
                            Width  : in     Base_Types.Width_T)
         is abstract;
   function Pen_Width (Object : Object_T) return Base_Types.Width_T
         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;
package body Line_Draw is

   procedure Print (Object : Object_T'Class) is
      Lines : constant Base_Types.Lines_T := Object.Convert;
   begin
      for Index in Lines'Range loop
         Ada.Text_IO.Put_Line ("Line" & Index'Image);
         Ada.Text_IO.Put_Line
           ("  From: " & Base_Types.Image (Lines (Index) (1)));
         Ada.Text_IO.Put_Line
           ("    To: " & Base_Types.Image (Lines (Index) (2)));
      end loop;
   end Print;

end Line_Draw;
```

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_Width (Object : in out Object_T;
                            Width  :        Base_Types.Width_T);
   function Pen_Width (Object : Object_T) return Base_Types.Width_T;
private
   type Object_T is
   abstract new Geometry.Object_T and Line_Draw.Object_T with record
      The_Color     : Base_Types.Color_T := 0;
      The_Pen_Width : Base_Types.Width_T := 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.The_Color := Color;
   end Set_Color;
   function Color (Object : Object_T) return Base_Types.Color_T is (Object.The_Color);

   procedure Set_Pen_Width (Object : in out Object_T;
                            Width  :        Base_Types.Width_T) is
   begin
      Object.The_Pen_Width := Width;
   end Set_Pen_Width;
   function Pen_Width (Object : Object_T) return Base_Types.Width_T is (Object.The_Pen_Width);
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

Exceptions

Introduction

Rationale for Exceptions

- Textual separation from normal processing
- Rigorous Error Management
  - Cannot be ignored, unlike status codes from routines
  - Example: running out of gasoline in an automobile

```
package Automotive is
  type Vehicle is record
    Fuel_Quantity, Fuel_Minimum : Float;
    Oil_Temperature : Float;
    ...
  end record;
  Fuel_Exhausted : exception;
  procedure Consume_Fuel (Car : in out Vehicle);
  ...
end Automotive;
```

Semantics Overview

- Exceptions become active by being *raised*
  - Failure of implicit language-defined checks
  - Explicitly by application
- Exceptions occur at run-time
  - A program has no effect until executed
- May be several occurrences active at same time
  - One per task of control
- Normal execution abandoned when they occur
  - Error processing takes over in response
  - Response specified by *exception handlers*
  - *Handling the exception* means taking action in response
  - Other tasks need not be affected

Semantics Example: Raising

```
package body Automotive is
  function Current_Consumption return Float is
    ...
  end Current_Consumption;
  procedure Consume_Fuel (Car : in out Vehicle) is
  begin
    if Car.Fuel_Quantity <= Car.Fuel_Minimum then
      raise Fuel_Exhausted;
    else -- decrement quantity
      Car.Fuel_Quantity := Car.Fuel_Quantity -
                           Current_Consumption;
    end if;
  end Consume_Fuel;
  ...
end Automotive;
```

Semantics Example: Handling

```
procedure Joy_Ride is
  Hot_Rod : Automotive.Vehicle;
  Bored : Boolean := False;
  use Automotive;
begin
  while not Bored loop
    Steer_Aimlessly (Bored);
    -- error situation cannot be ignored
    Consume_Fuel (Hot_Rod);
  end loop;
  Drive_Home;
exception
  when Fuel_Exhausted =>
    Push_Home;
end Joy_Ride;
```

Handler Part Is Skipped Automatically

- If no exceptions are active, returns normally

```
begin
   ...
-- if we get here, skip to end
exception
   when Name1 =>
   ...
   when Name2 | Name3 =>
   ...
   when Name4 =>
   ...
end;
```

Handlers

Exception Handler Part

- Contains the exception handlers within a frame
  - Within block statements, subprograms, tasks, etc.
- Separates normal processing code from abnormal
- Starts with the reserved word `exception`
- Optional

```
begin
  sequence_of_statements
[ exception
    exception_handler
    { exception handler } ]
end
```

Exception Handlers Syntax

- Associates exception names with statements to execute in response
- If used, **others** must appear at the end, by itself
  - Associates statements with all other exceptions
- Syntax

```
exception_handler ::=
  when exception_choice { | exception_choice } =>
    sequence_of_statements
exception_choice ::= exception_name | others
```

Similarity To Case Statements

- Both structure and meaning
- Exception handler

```
...
exception
  when Constraint_Error | Storage_Error | Program_Error =>
  ...
  when others =>
  ...
end;
```

- Case statement

```
case exception_name is
  when Constraint_Error | Storage_Error | Program_Error =>
  ...
  when others =>
  ...
end case;
```

Handlers Don't "Fall Through"

```
begin
  ...
  raise Name3;
  -- code here is not executed
  ...
exception
  when Name1 =>
     -- not executed
     ...
  when Name2 | Name3 =>
     -- executed
     ...
  when Name4 =>
     -- not executed
     ...
end;
```

When An Exception Is Raised

- Normal processing is abandoned
- Handler for active exception is executed, if any
- Control then goes to the caller
- If handled, caller continues normally, otherwise repeats the above

- Caller

```
...
Joy_Ride;
Do_Something_At_Home;
...
```

- Callee

```
procedure Joy_Ride is
   ...
begin
   ...
   Drive_Home;
exception
   when Fuel_Exhausted =>
      Push_Home;
end Joy_Ride;
```

Handling Specific Statements' Exceptions

```
begin
  loop
    Prompting : loop
      Put (Prompt);
      Get_Line (Filename, Last);
      exit when Last > Filename'First - 1;
    end loop Prompting;
    begin
      Open (F, In_File, Filename (1..Last));
      exit;
    exception
      when Name_Error =>
        Put_Line ("File '" & Filename (1..Last) &
                  "' was not found.");
    end;
  end loop;
```

Exception Handler Content

- No restrictions
  - Block statements, subprogram calls, etc.
- Do whatever makes sense

```
begin
  ...
exception
  when Some_Error =>
    declare
      New_Data : Some_Type;
    begin
      P (New_Data);
      ...
    end;
end;
```

Quiz

```
procedure Main is
   A, B, C, D : Integer range 0 .. 100;
begin
   A := 1; B := 2; C := 3; D := 4;
   begin
      D := A - C + B;
   exception
      when others => Put_Line ("One");
                     D := 1;
   end;
   D := D + 1;
   begin
      D := D / (A - C + B);
   exception
      when others => Put_Line ("Two");
                     D := -1;
   end;
exception
   when others =>
      Put_Line ("Three");
end Main;
```

What will get printed?

A. One, Two, Three
B. Two, Three
C. Two
D. Three

Quiz

```
1  procedure Main is
2     A, B, C, D : Integer range 0 .. 100;
3  begin
4     A := 1; B := 2; C := 3; D := 4;
5     begin
6        D := A - C + B;
7     exception
8        when others => Put_Line ("One");
9                       D := 1;
10    end;
11    D := D + 1;
12    begin
13       D := D / (A - C + B);
14    exception
15       when others => Put_Line ("Two");
16                      D := -1;
17    end;
18 exception
19    when others =>
20       Put_Line ("Three");
21 end Main;
```

What will get printed?

A. One, Two, Three
B. ***Two, Three***
C. Two
D. Three

Explanations

A. Although `(A - C)` is not in the range of `natural`, the range is only checked on assignment, which is after the addition of `B`, so `One` is never printed
B. Correct
C. If we reach `Two`, the assignment on line 16 will cause `Three` to be reached
D. Divide by 0 on line 13 causes an exception, so `Two` must be called

Implicitly and Explicitly Raised Exceptions

Implicitly-Raised Exceptions

- Correspond to language-defined checks
- Can happen by statement execution

```
K := -10;  -- where K must be greater than zero
```

- Can happen by declaration elaboration

```
Doomed : array (Positive) of Big_Type;
```

Some Language-Defined Exceptions

- `Constraint_Error`
  - Violations of constraints on range, index, etc.
- `Program_Error`
  - Runtime control structure violated (function with no return ...)
- `Storage_Error`
  - Insufficient storage is available
- For a complete list see RM Q-4

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

```
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;
```

User-Defined Exceptions

User-Defined Exceptions

- Syntax

```
defining_identifier_list : exception;
```

- Behave like predefined exceptions
  - Scope and visibility rules apply
  - Referencing as usual
  - Some minor differences
- Exception identifiers' use is restricted
  - `raise` statements
  - Handlers
  - Renaming declarations

User-Defined Exceptions Example

- An important part of the abstraction
- Designer specifies how component can be used

```
package Stack is
  Underflow, Overflow : exception;
  procedure Push (Item : in Integer);
  ...
end Stack;

package body Stack is
  procedure Push (Item : in Integer) is
  begin
    if Top = Index'Last then
      raise Overflow;
    end if;
    Top := Top + 1;
    Values (Top) := Item;
  end Push;
...
```

Propagation

Propagation

- Control does not return to point of raising
  - Termination Model
- When a handler is not found in a block statement
  - Re-raised immediately after the block
- When a handler is not found in a subprogram
  - Propagated to caller at the point of call
- Propagation is dynamic, back up the call chain
  - Not based on textual layout or order of declarations
- Propagation stops at the main subprogram
  - Main completes abnormally unless handled

Propagation Demo

```
procedure Do_Something is
  Error : exception;
  procedure Unhandled is
  begin
    Maybe_Raise(1);
  end Unhandled;
  procedure Handled is
  begin
    Unhandled;
    Maybe_Raise(2);
  exception
    when Error =>
      Print("Handle 1 or 2");
  end Handled;
```

```
begin -- Do_Something
  Maybe_Raise(3);
  Handled;
exception
  when Error =>
    Print("Handle 3");
end Do_Something;
```

Termination Model

- When control goes to handler, it continues from here

```
procedure Joy_Ride is
begin
   loop
      Steer_Aimlessly;

      -- If next line raises Fuel_Exhausted, go to handler
      Consume_Fuel;
   end loop;
exception
  when Fuel_Exhausted => -- Handler
    Push_Home;
    -- Resume from here: loop has been exited
end Joy_Ride;
```

Quiz

```
2  Main_Problem : exception;
3  I : Integer;
4  function F (P : Integer) return Integer is
5  begin
6    if P > 0 then
7      return P + 1;
8    elsif P = 0 then
9      raise Main_Problem;
10   end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error    => Put_Line ("Program Error");
18   when others           => Put_Line ("Unknown problem");
```

What will get printed if `Input_Value` on line 13 is `Integer'Last`?

A. `Unknown Problem`
B. `Success`
C. `Constraint Error`
D. `Program Error`

Quiz

```
2  Main_Problem : exception;
3  I : Integer;
4  function F (P : Integer) return Integer is
5  begin
6    if P > 0 then
7      return P + 1;
8    elsif P = 0 then
9      raise Main_Problem;
10   end if;
11 end F;
12 begin
13   I := F(Input_Value);
14   Put_Line ("Success");
15 exception
16   when Constraint_Error => Put_Line ("Constraint Error");
17   when Program_Error    => Put_Line ("Program Error");
18   when others           => Put_Line ("Unknown problem");
```

What will get printed if `Input_Value` on line 13 is `Integer'Last`?

A. `Unknown Problem`
B. `Success`
C. *`Constraint Error`*
D. `Program Error`

Explanations

A. `"Unknown Problem"` is printed by the **when others** due to the raise on line 9 when `P` is 0
B. `"Success"` is printed when 0 < `P` < `Integer'Last`
C. Trying to add 1 to `P` on line 7 generates a `Constraint_Error`
D. `Program_Error` will be raised by `F` if `P` < 0 (no **return** statement found)

Exceptions as Objects

Exceptions Are Not Objects

- May not be manipulated
  - May not be components of composite types
  - May not be passed as parameters
- Some differences for scope and visibility
  - May be propagated out of scope

But You Can Treat Them As Objects

- For raising and handling, and more
- Standard Library

```
package Ada.Exceptions is
  type Exception_Id is private;
  procedure Raise_Exception (E : Exception_Id;
                             Message : String := "");
  ...
  type Exception_Occurrence is limited private;
  function Exception_Name (X : Exception_Occurrence)
      return String;
  function Exception_Message (X : Exception_Occurrence)
      return String;
  function Exception_Information (X : Exception_Occurrence)
      return String;
  procedure Reraise_Occurrence (X : Exception_Occurrence);
  procedure Save_Occurrence (
    Target : out Exception_Occurrence;
    Source : Exception_Occurrence);
  ...
end Ada.Exceptions;
```

Exception Occurrence

- Syntax associates an object with active exception

```
when defining_identifier : exception_name ... =>
```

- A constant view representing active exception
- Used with operations defined for the type

```
exception
  when Caught_Exception : others =>
    Put (Exception_Name (Caught_Exception));
```

Exception_Occurrence Query Functions

- **Exception_Name**
  - Returns full expanded name of the exception in string form
    - Simple short name if space-constrained
  - Predefined exceptions appear as just simple short name
- **Exception_Message**
  - Returns string value specified when raised, if any
- **Exception_Information**
  - Returns implementation-defined string content
  - Should include both exception name and message content
  - Presumably includes debugging information
    - Location where exception occurred
    - Language-defined check that failed (if such)

Exception ID

- For an exception identifier, the *identity* of the exception is `<name>'Identity`

```
Mine : exception
use Ada.Exceptions;
...
exception
   when Occurrence : others =>
      if Exception_Identity(Occurrence) = Mine'Identity
      then
         ...
```

Raise Expressions

Raise Expressions

Ada 2012

- **Expression** raising specified exception **at run-time**

```
Foo : constant Integer := (case X is
                            when 1 => 10,
                            when 2 => 20,
                            when others => raise Error);
```

In Practice

Exceptions Are Not Always Appropriate

- What does it mean to have an unexpected error in a safety-critical application?
  - Maybe there's no reasonable response

Relying On Exception Raising Is Risky

- They may be **suppressed**
  - By runtime environment
  - By build switches
- Not recommended

```
function Tomorrow (Today : Days) return Days is
begin
  return Days'Succ (Today);
exception
  when Constraint_Error =>
    return Days'First;
end Tomorrow;
```

- Recommended

```
function Tomorrow (Today : Days) return Days is
begin
  if Today = Days'Last then
    return Days'First;
  else
    return Days'Succ (Today);
  end if;
end Tomorrow;
```

Lab

Exceptions Lab

(Simplified) Input Verifier

- Overview
  - Create an application that converts strings to numeric values
- Requirements
  - Create a package to define your numeric type
  - Define a primitive to convert a string to your numeric type
    - The primitive should raise your own exceptions; one for out-of-range and one for illegal string
  - Main program should run multiple tests on the primitive

Exceptions Lab Solution - Numeric Types

```
package Numeric_Types is
   Illegal_String : exception;
   Out_Of_Range   : exception;

   Max_Int : constant := 2**15;
   type Integer_T is range -(Max_Int) .. Max_Int - 1;

   function Value (Str : String) return Integer_T;
end Numeric_Types;

package body Numeric_Types is

   function Legal (C : Character) return Boolean is
   begin
      return
        C in '0' .. '9' or C = '+' or C = '-' or C = '_' or C = 'e' or C = 'E';
   end Legal;

   function Value (Str : String) return Integer_T is
   begin
      for I in Str'Range loop
         if not Legal (Str (I)) then
            raise Illegal_String;
         end if;
      end loop;
      return Numeric_Types.Integer_T'Value (Str);
   exception
      when Constraint_Error =>
         raise Out_Of_Range;
   end Value;

end Numeric_Types;
```

Exceptions Lab Solution - Main

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

   procedure Print_Value (Str : String) is
      Value : Numeric_Types.Integer_T;
   begin
      Ada.Text_IO.Put (Str & " => ");
      Value := Numeric_Types.Value (Str);
      Ada.Text_IO.Put_Line (Numeric_Types.Integer_T'Image (Value));
   exception
      when Numeric_Types.Out_Of_Range =>
         Ada.Text_IO.Put_Line ("Out of range");
      when Numeric_Types.Illegal_String =>
         Ada.Text_IO.Put_Line ("Illegal entry");
   end Print_Value;

begin
   Print_Value ("123");
   Print_Value ("2_3_4");
   Print_Value ("-345");
   Print_Value ("+456");
   Print_Value ("1234567890");
   Print_Value ("123abc");
   Print_Value ("12e3");
end Main;
```

Summary

Summary

- Should be for unexpected errors
- Give clients the ability to avoid them
- If handled, caller should see normal effect
  - Mode `out` parameters assigned
  - Function return values provided
- Package **Ada.Exceptions** provides views as objects
  - For both raising and special handling
  - Especially useful for debugging
- Checks may be suppressed

Advanced Tasking

Introduction

A Simple Task

- Parallel code execution via **task**
- `limited` types (No copies allowed)

```
procedure Main is
   task type 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 ("receive " & 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**

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 type 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 type 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

```
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;
```

Which completion(s) of P is(are) 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;
```

Which completion(s) of `P` is(are) 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 type T;
end P;

package body P is
   task body T is
      loop
         delay 1.0;
         Put_Line ("tick");
      end loop;
   end T;

   Task_Instance : 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 : " & V);
  end Receive_Message;
or
  accept Stop;
    exit;
  end select;
```

Guard Conditions

- `accept` may depend on a **guard condition** with `when`
  - Evaluated when entering `select`
- May use a *guard condition*, that **only** accepts entries on a **boolean** condition
  - Condition is evaluated when the task reaches it

```
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* which is evaluated when
  - A task calls an `entry`
  - A protected `entry` or `procedure` is **exited**
- Several tasks can be waiting on the same `entry`
  - Only **one** may 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;
```

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;
```

Quiz

```
procedure Main is
    protected type O is
       entry P;
    private
        Ok : Boolean := False;
    end O;

    protected body O is
       entry P when not Ok is
       begin
          Ok := True;
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. `Ok = True`
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
procedure Main is
    protected type O is
       entry P;
    private
        Ok : Boolean := False;
    end O;

    protected body O is
       entry P when not Ok is
       begin
          Ok := True;
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. `Ok = True`
B. Nothing
C. ***Compilation error***
D. Runtime error

`O` is a **`protected type`**, needs instantiation

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 : " & V);
  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;
```

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 type 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 type T;

   task body 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

```ada
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

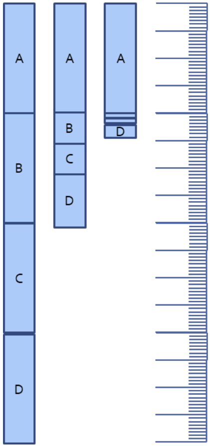
```
type T1 is range 1 .. 4;
for T1'Size use 4;
for T1'Alignment use 8;

-- 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

- Exact mapping between a record and its binary representation
- Optimization purposes, or hardware requirements
  - Driver mapped on the address space, communication protocol...
- Fields represented as
  `<name> at <byte> range <starting-bit> .. <ending-bit>`

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

Array Representation Clauses

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

```
type 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

- **Bit_Order** for a type's endianness
- **Scalar_Storage_Order** for composite types
  - Endianess of components' ordering
  - GNAT-specific
  - Must be consistent with **Bit_Order**
- Compiler will peform needed bitwise transformations when performing operations

```
type Rec is record
   A : Integer;
   B : Boolean;
end record;
for Rec use record
   A at 0 range 0 .. 31;
   B at 0 range 32 .. 33;
end record;
for Rec'Bit_Order use System.High_Order_First;
for Rec'Scalar_Storage_Order use System.High_Order_First;

-- using Ada 2012 aspects
type Ar is array (1 .. 1000) of Boolean with
  Scalar_Storage_Order => System.Low_Order_First;
```

Change of Representation

- Explicit new type can be used to set 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
```

- The exact sequence of reads and writes from the source code must appear in the generated code
  - No optimization of reads and writes
- Volatile types are passed by-reference

Ada Address Example

```
type Bitfield is array (Integer range <>) of Boolean;
pragma Component_Size (1);

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*: multiple objects are accessing the same address
    - Types can be different
    - Two pointers pointing to the same address
    - Two references onto the same address
    - Two objects 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
  - Many compilers will warn if the type sizes do not match

Inline Assembly

Calling Assembly Code

- Calling assembly code is a vendor-specific extension
- GNAT allows passing assembly with **System.Machine_Code.ASM**
  - Handled by the linker directly
- 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);
```

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

- You can group several instructions

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

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

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
g	global (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 your toolchain's `as` assembler manual for syntax

Volatile and Clobber ASM Parameters

- `Volatile` → `True` deactivates optimizations with regards to suppressed instructions
- `Clobber` → `"reg1, reg2, ..."` contains the list of registers considered to be "destroyed" by the use of the ASM call
  - `memory` if the memory is accessed
    - Compiler won't use memory cache in registers across the instruction
  - `cc` if flags might have changed

Instruction Counter Example (x86)

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

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