Introduction

About AdaCore

The Company

- Founded in 1994
- Centered around helping developers build **safe, secure and reliable** software
- Headquartered in New York and Paris
  - Representatives in countries around the globe
- Roots in Open Source software movement
  - GNAT compiler is part of GNU Compiler Collection (GCC)

About This Training

Your Trainer

- Experience in software development
  - Languages
  - Methodology
- Experience teaching this class

Goals of the training session

- What you should know by the end of the training
- Syllabus overview
  - The syllabus is a guide, but we might stray off of it
  - ...and that's OK: we're here to cover **your needs**

Roundtable

- 5 minute exercise
  - Write down your answers to the following
  - Then share it with the room
- Experience in software development
  - Languages
  - Methodology
- Experience and interest with the syllabus
  - Current and upcoming projects
  - Curious for something?
- Your personal goals for this training
  - What do you want to have coming out of this?
- Anecdotes, stories... feel free to share!
  - Most interesting or funny bug you've encountered?
  - Your own programming interests?

Course Presentation

- Slides
- Quizzes
- Labs
  - Hands-on practice
  - Recommended setup: latest GNAT Studio
  - Class reflection after some labs
- Demos
  - Depending on the context
- Daily schedule

Styles

- *This* is a definition
- **`this/is/a.path`**
- `code is highlighted`
- `commands are emphasised --like-this`

> ⚠ **Warning**
>
> This is a warning

> **Note**
>
> This is an important piece of info

> **Tip**
>
> This is a tip

Basic Types

Introduction

Strong Typing

- Definition of *type*
  - Applicable **values**
  - Applicable *primitive* **operations**
- Compiler-enforced
  - **Check** of values and operations
  - Easy for a computer

Tip

Developer can focus on **earlier** phase: requirement

Strongly-Typed Vs Weakly-Typed Languages

- Weakly-typed:
  - Conversions are **unchecked**
  - Type errors are easy

```
typedef enum {north, south, east, west} direction;
typedef enum {sun, mon, tue, wed, thu, fri, sat} days;
direction heading = north;

heading = 1 + 3 * south/sun;// what?
```

- Strongly-typed:
  - Conversions are **checked**
  - Type errors are hard

```
type Directions is (North, South, East, West);
type Days is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
Heading : Directions := North;
...
Heading := 1 + 3 * South/Sun; --  Compile Error
```

A Little Terminology

- **Declaration** creates a **type name**

```
type <name> is <type definition>;
```

- **Type-definition** defines its structure
  - Characteristics, and operations
  - Base "class" of the type

```
type Type_1 is digits 12; -- floating-point
type Type_2 is range -200 .. 200; -- signed integer
type Type_3 is mod 256; -- unsigned integer
```

- *Representation* is the memory-layout of an **object** of the type

Abstract Data Types (ADT)

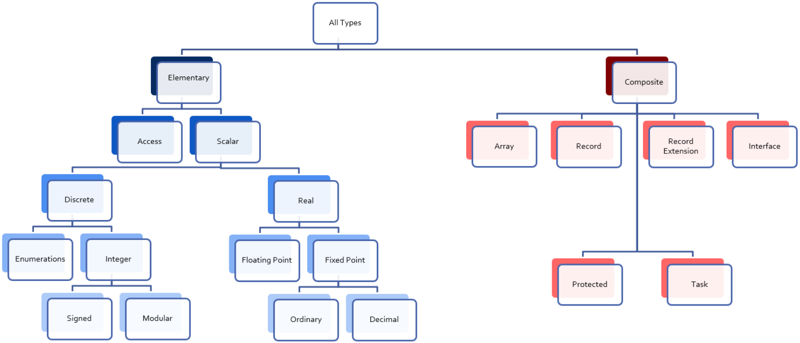
- **Variables** of the **type** encapsulate the **state**
- Classic definition of an ADT
  - Set of **values**
  - Set of **operations**
  - **Hidden** compile-time **representation**
- Compiler-enforced
  - Check of values and operation
  - Easy for a computer
  - Developer can focus on **earlier** phase: requirements

Ada "Named Typing"

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

```
type Yen is range 0 .. 100_000_000;
type Ruble is range 0 .. 100_000_000;
Mine : Yen;
Yours : Ruble;
...
Mine := Yours; -- not legal
```

Categories of Types



Scalar Types

- Indivisible: No *components* (also known as *fields* or *elements*)
- **Relational** operators defined (<, =, ...)
  - **Ordered**
- Have common **attributes**
- **Discrete** Types
  - Integer
  - Enumeration
- **Real** Types
  - Floating-point
  - Fixed-point

Discrete Types

- **Individual** ("discrete") values
  - 1, 2, 3, 4 ...
  - Red, Yellow, Green
- Integer types
  - Signed integer types
  - Modular integer types
    - Unsigned
    - **Wrap-around** semantics
    - Bitwise operations
- Enumeration types
  - Ordered list of **logical** values

Attributes

- Properties of entities that can be queried like a function
  - May take input parameters
- Defined by the language and/or compiler
  - Language-defined attributes found in RM K.2
  - *May* be implementation-defined
    - GNAT-defined attributes found in GNAT Reference Manual
  - Cannot be user-defined
- Attribute behavior is generally pre-defined
  - `Type_T'Digits` gives number of digits used in `Type_T` definition
- Some attributes can be modified by coding behavior
  - `Typemark'Size` gives the size of `Typemark`
    - Determined by compiler **OR** by using a representation clause
  - `Object'Image` gives a string representation of `Object`
    - Default behavior which can be replaced by aspect `Put_Image`
- Examples

```
J := Object'Size;
K := Array_Object'First(2);
```

Type Model Run-Time Costs

- Checks at compilation **and** run-time
- **Same performance** for identical programs
  - Run-time type checks can be disabled

> **Note**
> Compile-time check is *free*

C

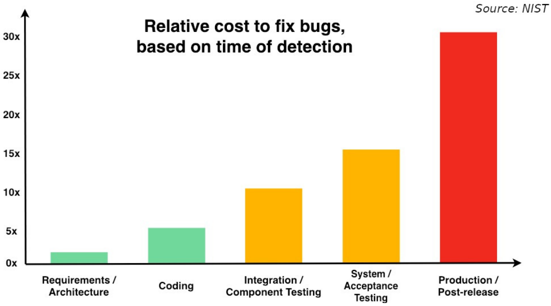
```c
int X;
int Y; // range 1 .. 10
...
if (X > 0 && X < 11)
  Y = X;
else
  // signal a failure
```

Ada

```ada
X : Integer;
Y, Z : Integer range 1 .. 10;
...
Y := X;
Z := Y; -- no check required
```

The Type Model Saves Money

- Shifts fixes and costs to **early phases**
- Cost of an error *during a flight*?



Discrete Numeric Types

Signed Integer Types

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

```
type &lt;identifier&gt; is range  &lt;lower&gt; .. &lt;upper&gt;;
```

- Implicit numeric operators

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

Signed Integer Bounds

- Must be **static**
  - Compiler selects **base type**
  - Hardware-supported integer type
  - Compilation **error** if not possible

Predefined Signed Integer Types

- `Integer` >= **16 bits** wide
- Other **probably** available
  - `Long_Integer`, `Short_Integer`, etc.
  - Guaranteed ranges: `Short_Integer <= Integer <= Long_Integer`
  - Ranges are all **implementation-defined**

⚠ **Warning**

Portability not guaranteed

- But usage may be difficult to avoid

Operators for Signed Integer Type

- By increasing precedence

relational operator = | /= | < | <= | > | >=
binary adding operator + | -
unary adding operator + | -
multiplying operator * | / | mod | rem
highest precedence operator ** | abs

Note

Exponentiation (**) result will be a signed integer

- So power **must** be `Integer` >= 0

Warning

Division by zero → `Constraint_Error`

Signed Integer Overflows

- Finite binary representation
- Common source of bugs

```
K : Short_Integer := Short_Integer'Last;
...
K := K + 1;

 2#0111_1111_1111_1111#  = (2**15)-1

+                     1

======================
 2#1000_0000_0000_0000#  = -32,768
```

Signed Integer Overflow: Ada Vs Others

- Ada
  - `Constraint_Error` standard exception
  - Incorrect numerical analysis
- Java
  - Silently **wraps** around (as the hardware does)
- C/C++
  - **Undefined** behavior (typically silent wrap-around)

String Attributes for All Scalars

- `T'Image (input)`
  - Converts `T` → `String`
- `T'Value (input)`
  - Converts `String` → `T`

```
Number : Integer := 12345;
Input  : String (1 .. N);
...
Put_Line (Integer'Image (Number));
...
Get (Input);
Number := Integer'Value (Input);
```

Range Attributes for All Scalars

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

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

Neighbor Attributes for All Scalars

- `T'Pred (Input)`
  - Predecessor of specified value
  - Input type must be T
- `T'Succ (Input)`
  - Successor of specified value
  - Input type must be T

```
type Signed_T is range -128 .. 127;
type Unsigned_T is mod 256;
Signed   : Signed_T := -1;
Unsigned : Unsigned_T := 0;
...
Signed := Signed_T'Succ (Signed); -- Signed = 0
...
Unsigned := Unsigned_T'Pred (Unsigned); -- Signed = 255
```

Min/Max Attributes for All Scalars

- T'Min (Value_A, Value_B)
  - **Lesser** of two T
- T'Max (Value_A, Value_B)
  - **Greater** of two T

```
Safe_Lower : constant := 10;
Safe_Upper : constant := 30;
C : Integer := 15;
...
C := Integer'Max (Safe_Lower, C - 1);
...
C := Integer'Min (Safe_Upper, C + 1);
```

Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;
C2 : constant := 2 ** 1024 + 10;
C3 : constant := C1 - C2;
V  : Integer := C1 - C2;
```

A. Compile error
B. Run-time error
C. V is assigned to -10
D. Unknown - depends on the compiler

Quiz

What happens when you try to compile/run this code?

```
C1 : constant := 2 ** 1024;
C2 : constant := 2 ** 1024 + 10;
C3 : constant := C1 - C2;
V  : Integer := C1 - C2;
```

A. Compile error
B. Run-time error
C. ***V is assigned to -10***
D. Unknown - depends on the compiler

Explanations

- 2^{1024} too big for most runtimes BUT
- `C1`, `C2`, and `C3` are named numbers, not typed constants
  - Compiler uses unbounded precision for named numbers
  - Large intermediate representation does not get stored in object code
- For assignment to `V`, subtraction is computed by compiler
  - `V` is assigned the value -10

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);   -- run-time 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 => Intrinsic;
```

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)`

Enumeration Types

Enumeration Types

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

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

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

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

Enumeration Type Operations

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

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

Character Types

- Literals
  - Enclosed in single quotes eg. `'A'`
  - Case-sensitive
- **Special-case** of enumerated type
  - At least one character enumeral
- System-defined `Character`
- Can be user-defined

```
type EBCDIC is (nul, ..., 'a' , ..., 'A', ..., del);
Control : EBCDIC := 'A';
Nullo : EBCDIC := nul;
```

Language-Defined Type Boolean

- Enumeration

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

- Supports assignment, relational operators, attributes

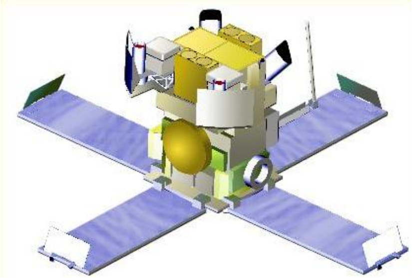
```
A : Boolean;
Counter : Integer;
...
A := (Counter = 22);
```

- Logical operators **and**, **or**, **xor**, **not**

```
A := B or (not C); -- For A, B, C boolean
```

Why Boolean Isn't Just an Integer?

- Example: Real-life error
  - HETE-2 satellite **attitude control** system software (ACS)
  - Written in **C**
- Controls four "solar paddles"
  - Deployed after launch



Why Boolean Isn't Just an Integer!

- **Initially** variable with paddles' state
  - Either **all** deployed, or **none** deployed
- Used `int` as a boolean

```
if (rom->paddles_deployed == 1)
  use_deployed_inertia_matrix();
else
  use_stowed_inertia_matrix();
```

- Later `paddles_deployed` became a **4-bits** value
  - One bit per paddle
  - `0` → none deployed, `0xF` → all deployed
- Then, `use_deployed_inertia_matrix()` if only first paddle is deployed!
- Better: boolean function `paddles_deployed()`
  - Single line to modify

Boolean Operators' Operand Evaluation

- Evaluation order **not specified**
- May be needed
  - Checking value **before** operation
  - Dereferencing null pointers
  - Division by zero

```
if Divisor /= 0 and K / Divisor = Max then ... -- Problem!
```

Short-Circuit Control Forms

- **Short-circuit** → **fixed** evaluation order
- Left-to-right
- Right only evaluated **if necessary**
  - `and then`: if left is `False`, skip right

    ```
    Divisor /= 0 and then K / Divisor = Max
    ```

  - `or else`: if left is `True`, skip right

    ```
    Divisor = 0 or else K / Divisor = Max
    ```

Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement(s) is (are) legal?

A. `V1 :  Enum_T := Enum_T'Value ("Able");`
B. `V2 :  Enum_T := Enum_T'Value ("BAKER");`
C. `V3 :  Enum_T := Enum_T'Value (" charlie ");`
D. `V4 : Enum_T := Enum_T'Value ("Able Baker Charlie");`

Quiz

```
type Enum_T is (Able, Baker, Charlie);
```

Which statement(s) is (are) legal?

A. `V1 :  Enum_T := Enum_T'Value ("Able");`
B. `V2 :  Enum_T := Enum_T'Value ("BAKER");`
C. `V3 :  Enum_T := Enum_T'Value (" charlie ");`
D. `V4 : Enum_T := Enum_T'Value ("Able Baker Charlie");`

Explanations

A. Legal
B. Legal - conversion is case-insensitive
C. Legal - leading/trailing blanks are ignored
D. Value tries to convert entire string, which will fail at run-time

Representation Values

Enumeration Representation Values

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

```
type Enum_T is (Able, Baker, Charlie, David);
for Enum_T use
  (Able => 3, Baker => 15, Charlie => 63, David => 255);
```

  - Enumerals are ordered **logically** (not by value)
- Prior to Ada 2022
  - Only way to get value is through `Unchecked_Conversion`

```
function Value is new Ada.Unchecked_Conversion
   (Enum_T, Integer_8);
I : Integer_8;

begin
   I := Value (Charlie);
```

- New attributes in Ada 2022
  - `'Enum_Rep` to get representation value

    `Charlie'Enum_Rep` → 63
  - `'Enum_Val` to convert integer to enumeral (if possible)

    `Enum_T'Enum_Val (15)` → `Baker`

    `Enum_T'Enum_Val (16)` → raise `Constraint_Error`

Order Attributes for All Discrete Types

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

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

Quiz

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

Which of the following proposition(s) are true?

A. `T'Value (V) = 1`
B. `T'Pos (V) = 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 component of `String`
  - Uses attributes `'Image` / `'Value`
- `Wide_Character`
  - 16-bit Unicode
  - Base component of `Wide_Strings`
  - Uses attributes `'Wide_Image` / `'Wide_Value`
- `Wide_Wide_Character`
  - 32-bit Unicode
  - Base component 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 run-time
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 run-time
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)`*

Real Types

Real Types

- Approximations to **continuous** values
  - 1.0, 1.1, 1.11, 1.111 ... 2.0, ...
  - Finite hardware → approximations
- Floating-point
  - **Variable** exponent
  - **Large** range
  - Constant **relative** precision
- Fixed-point
  - **Constant** exponent
  - **Limited** range
  - Constant **absolute** precision
  - Subdivided into Binary and Decimal
- Class focuses on floating-point

Real Type (Floating and Fixed) Literals

- **Must** contain a fractional part
- No silent promotion

```
type Phase is digits 8; -- floating-point
OK : Phase := 0.0;
Bad : Phase := 0 ; -- compile error
```

Declaring Floating Point Types

- Syntax

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

  - *digits* → **minimum** number of significant digits
  - **Decimal** digits, not bits

- Compiler choses representation
  - From **available** floating point types
  - May be **more** accurate, but not less
  - If none available → declaration is **rejected**

- `System.Max_Digits` - constant specifying maximum digits of precision available for runtime

```
type Very_Precise_T is digits System.Max_Digits;
```

Need to do `with System;` *to get visibility*

Predefined Floating Point Types

- Type `Float` `>= 6` digits
- Additional implementation-defined types
  - `Long_Float` `>= 11` digits
- General-purpose

Tip

It is best, and easy, to **avoid** predefined types

- To keep **portability**

Floating Point Type Operators

- By increasing precedence

relational operator `= | /= | < | >= | > | >=`
binary adding operator `+ | -`
unary adding operator `+ | -`
multiplying operator `* | /`
highest precedence operator `** | abs`

Note

Exponentiation (`**`) result will be real

- So power must be `Integer`
  - Not possible to ask for root
  - `X**0.5` → `sqrt (x)`

Floating Point Type Attributes

- *Core* attributes

```
type My_Float is digits N;  -- N static
```

  - `My_Float'Digits`
    - Number of digits **requested** (N)
  - `My_Float'Base'Digits`
    - Number of **actual** digits
  - `My_Float'Rounding (X)`
    - Integral value nearest to `X`
    - *Note:* `Float'Rounding (0.5) = 1` and `Float'Rounding (-0.5) = -1`
- Model-oriented attributes
  - Advanced machine representation of the floating-point type
  - Mantissa, strict mode

Numeric Types Conversion

- Ada's integer and real are *numeric*
  - Holding a numeric value
- Special rule: can always convert between numeric types
  - Explicitly

Warning

`Float` → `Integer` causes **rounding**

```
declare
   N : Integer := 0;
   F : Float := 1.5;
begin
   N := Integer (F); -- N = 2
   F := Float (N); -- F = 2.0
```

Quiz

What is the output of this code?

```
declare
   F : Float := 7.6;
   I : Integer := 10;
begin
   F := Float (Integer (F) / I);
   Put_Line (Float'Image (F));
end;
```

A. 7.6E-01
B. Compile Error
C. 8.0E-01
D. 0.0

Quiz

What is the output of this code?

```
declare
   F : Float := 7.6;
   I : Integer := 10;
begin
   F := Float (Integer (F) / I);
   Put_Line (Float'Image (F));
end;
```

A. 7.6E-01
B. Compile Error
C. 8.0E-01
D. ***0.0***

Explanations

A. Result of `F := F / Float (I);`
B. Result of `F := F / I;`
C. Result of `F := Float (Integer (F)) / Float (I);`
D. Integer value of `F` is 8. Integer result of dividing that by 10 is 0. Converting to float still gives us 0

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 ($+\varepsilon$ / $-\varepsilon$)

Miscellaneous

Checked Type Conversions

- Between "closely related" types
  - Numeric types
  - Inherited types
  - Array types
- Illegal conversions **rejected**
  - Unsafe **Unchecked_Conversion** available
- Called as if it was a function
  - Named using destination type name

    ```
    Target_Float := Float (Source_Integer);
    ```

  - Implicitly defined
  - **Must** be explicitly called

Default Value

- Not defined by language for **scalars**
- Can be done with an **aspect clause**
  - Only during type declarations
  - <value> must be static

```
type Type_Name is <type_definition>
     with Default_Value => <value>;
```

- Example

```
type Tertiary_Switch is (Off, On, Neither)
   with Default_Value => Neither;
Implicit : Tertiary_Switch; -- Implicit = Neither
Explicit : Tertiary_Switch := Neither;
```

Simple Static Type Derivation

- New type from an existing type
  - **Limited** form of inheritance: operations
  - **Not** fully OOP
  - More details later
- Strong type benefits
  - Only **explicit** conversion possible
  - eg. `Meters` can't be set from a `Feet` value
- Syntax

```
type identifier is new Base_Type [<constraints>]
```

- Example

```
type Measurement is digits 6;
type Distance is new Measurement
      range 0.0 .. Measurement'Last;
```

Subtypes

Subtype

- May **constrain** an existing type
- Still the **same** type
- Syntax

```
subtype Defining_Identifier is Type_Name [constraints];
```

  - `Type_Name` is an existing `type` or `subtype`

Note

If no constraint → type alias

Subtype Example

- Enumeration type with **range** constraint

```
type Days is (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
subtype Weekdays is Days range Mon .. Fri;
Workday : Weekdays; -- type Days limited to Mon .. Fri
```

- Equivalent to **anonymous** subtype

```
Same_As_Workday : Days range Mon .. Fri;
```

Kinds of Constraints

- Range constraints on scalar types

```
subtype Positive is Integer range 1 .. Integer'Last;
subtype Natural is Integer range 0 .. Integer'Last;
subtype Weekdays is Days range Mon .. Fri;
subtype Symmetric_Distribution is
    Float range -1.0 .. +1.0;
```

- Other kinds, discussed later
- Constraints apply only to values
- Representation and set of operations are **kept**

Subtype Constraint Checks

- Constraints are checked
  - At initial value assignment
  - At assignment
  - At subprogram call
  - Upon return from subprograms
- Invalid constraints
  - Will cause `Constraint_Error` to be raised
  - May be detected at compile time
    - If values are **static**
    - Initial value → error
    - ... else → warning

```
Max : Integer range 1 .. 100 := 0; -- compile error
...
Max := 0; -- run-time error
```

Performance Impact of Constraints Checking

- Constraint checks have run-time performance impact
- The following code

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

- Generates assignment checks similar to

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

- These checks can be disabled with `-gnatp`

Optimizations of Constraint Checks

- Checks happen only if necessary
- Compiler assumes variables to be **initialized**
- So this code generates **no check**

```
procedure Demo is
  P, K : Integer range 0 .. 100;
begin
  P := K;
  -- But K is not initialized!
```

Range Constraint Examples

```
subtype Proper_Subset is Positive range 1 .. 10;
subtype Same_Constraints is Positive
    range 1 .. Integer'Last;
subtype Letter is Character range 'A' .. 'z';
subtype Upper_Case is Letter range 'A' .. 'Z';
subtype Lower_Case is Letter range 'a' .. 'z';
subtype Null_Range is Integer
    range 1 .. 0;  -- silly when hard-coded...
-- evaluated when subtype defined, not when object declared
subtype Dynamic is Integer range Lower .. Upper;
```

Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

A. `subtype A is Enum_Sub_T range Enum_Sub_T'Pred (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
B. `subtype B is range Sat .. Mon;`
C. `subtype C is Integer;`
D. `subtype D is digits 6;`

Quiz

```
type Enum_T is (Sat, Sun, Mon, Tue, Wed, Thu, Fri);
subtype Enum_Sub_T is Enum_T range Mon .. Fri;
```

Which subtype definition is valid?

A. `subtype A is Enum_Sub_T range Enum_Sub_T'Pred (Enum_Sub_T'First) .. Enum_Sub_T'Last;`
B. `subtype B is range Sat .. Mon;`
C. *`subtype C is Integer;`*
D. `subtype D is digits 6;`

Explanations

A. This generates a run-time error because the first enumeral specified is not in the range of `Enum_Sub_T`
B. Compile error - no type specified
C. Correct - standalone subtype
D. `Digits 6` is used for a type definition, not a subtype

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);
```

Default Values and Option Types

- 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
```

Tip

Using a meaningless value (`Neither`) to extend the range of the type is turning it into an *option type*. This idiom is very rich and allows for e.g. "in-flow" errors handling.

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; -- run-time error
Hue : Rainbow := Rainbow'Succ (Blue);      -- run-time error
```

Valid attribute

- The_Type'Valid is a Boolean
- True → the current representation for the given scalar is valid

```
procedure Main is
   subtype Small_T is Integer range 1 .. 3;
   Big   : aliased Integer := 0;
   Small : Small_T with Address => Big'Address;
begin
   for V in 0 .. 5 loop
      Big := V;
      Put_Line (Big'Image & " => " & Boolean'Image (Small'Valid));
   end loop;
end Main;
```

```
0 => FALSE
1 => TRUE
2 => TRUE
3 => TRUE
4 => FALSE
5 => FALSE
```

Idiom: Extended Ranges

- `Count` / `Positive_Count`
  - Sometimes as `Type_Ext` (extended) / `Type`
  - For counting vs indexing
    - An index goes from 1 to max length
    - A count goes from 0 to max length

```
-- ARM A.10.1
package Text_IO is
   ...
   type Count is range 0 .. implementation-defined;
   subtype Pos_Count is Count range 1 .. Count'Last;
```

Idiom: Partition

- Useful for splitting-up large enums

⚠ Warning

Be careful about checking that the partition is complete when items are added/removed.
With a `case`, the compiler automatically checks that for you.

Tip

Can have non-consecutive values with the `Predicate` aspect.

```
type Commands_T is (Lights_On, Lights_Off, Read, Write, Accelerate, Stop);
--  Complete partition of the commands
subtype IO_Commands_T is Commands_T range Read .. Write;
subtype Lights_Commands_T is Commands_T range Lights_On .. Lights_Off;
subtype Movement_Commands_T is Commands_T range Accelerate .. Stop;

subtype Physical_Commands_T is Commands_T
   with Predicate => Physical_Commands_T in Lights_Commands_T | Movement_Commands_T;

procedure Execute_Light_Command (C : Lights_Commands_T);

procedure Execute_Command (C : Commands_T) is
begin
   case C in --  partition must be exhaustive
      when Lights_Commands_T => Execute_Light_Command (C);
...
```

Idiom: Subtypes as Local Constraints

- Can replace defensive code
- Can be very useful in some identified cases
- Subtypes accept dynamic bounds, unlike types
- Checks happen through type-system
  - Can be disabled with `-gnatp`, unlike conditionals
  - Can also be a disadvantage

  ⚠ Warning
  Do not use for checks that should **always** happen, even in production.

- Constrain input range

```
subtype Incrementable_Integer is Integer range Integer'First .. Integer'Last - 1;
function Increment (I : Incrementable_Integer) return Integer;
```

- Constrain output range

```
subtype Valid_Fingers_T is Integer range 1 .. 5;
Fingers : Valid_Fingers_T := Prompt_And_Get_Integer ("Give me the number of a finger");
```

- Constrain array index

```
procedure Read_Index_And_Manipulate_Char (S : String) is
   subtype S_Index is Positive range S'Range;
   I : constant S_Index := Read_Positive;
   C : Character renames S (I);
```

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 run-time
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 run-time
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 run-time
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 run-time***
D. None, this does not compile

Lab

Basic Types Lab

- Create types to handle the following concepts
  - Determining average test score
    - Number of tests taken
    - Total of all test scores
  - Number of degrees in a circle
  - Collection of colors
- Create objects for the types you've created
  - Assign initial values to the objects
  - Print the values of the objects
- Modify the objects you've created and print the new values
  - Determine the average score for all the tests
  - Add 359 degrees to the initial circle value
  - Set the color object to the value right before the last possible value

Using the "Prompts" Directory

- Course material should have a link to a `Prompts` folder
- Folder contains everything you need to get started on the lab
  - GNAT Studio project file `default.gpr`
  - Annotated / simplified source files
    - Source files are templates for lab solutions
    - Files compile as is, but don't implement the requirements
    - Comments in source files give hints for the solution
- To load prompt, either
  - From within GNAT Studio, select File → Open Project and navigate to and open the appropriate `default.gpr` **OR**
  - From a command prompt, enter

    `gnatstudio -P <full path to GPR file>`
    - If you are in the appropriate directory, and there is only one GPR file, entering `gnatstudio` will start the tool and open that project
- These prompt folders should be available for most labs

Basic Types Lab Hints

- Understand the properties of the types
  - Do you need fractions or just whole numbers?
  - What happens when you want the number to wrap?
- Predefined package **Ada.Text_IO** is handy...
  - Procedure **Put_Line** takes a **String** as the parameter
- Remember attribute **'Image** returns a **String**

```
<typemark>'Image (Object)
Object'Image
```

Basic Types Extra Credit

- See what happens when your data is invalid / illegal
    - Number of tests = 0
    - Assign a very large number to the test score total
    - Color type only has one value
    - Add a number larger than 360 to the circle value

Basic Types Lab Solution - Declarations

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

   type Number_Of_Tests_T is range 0 .. 100;
   type Test_Score_Total_T is digits 6 range 0.0 .. 10_000.0;

   type Degrees_T is mod 360;

   type Cymk_T is (Cyan, Magenta, Yellow, Black);

   Number_Of_Tests   : Number_Of_Tests_T;
   Test_Score_Total : Test_Score_Total_T;

   Angle : Degrees_T;

   Color : Cymk_T;
```

Basic Types Lab Solution - Implementation

```
begin

   -- assignment
   Number_Of_Tests  := 15;
   Test_Score_Total := 1_234.5;
   Angle            := 180;
   Color            := Magenta;

   Put_Line (Number_Of_Tests'Image);
   Put_Line (Test_Score_Total'Image);
   Put_Line (Angle'Image);
   Put_Line (Color'Image);

   -- operations / attributes
   Test_Score_Total := Test_Score_Total / Test_Score_Total_T (Number_Of_Tests);
   Angle            := Angle + 359;
   Color            := Cymk_T'Pred (Cymk_T'Last);

   Put_Line (Test_Score_Total'Image);
   Put_Line (Angle'Image);
   Put_Line (Color'Image);

end Main;
```

Summary

Benefits of Strongly Typed Numerics

- **Prevent** subtle bugs
- Cannot mix `Apples` and `Oranges`
- Force to clarify **representation** needs
  - eg. constant with or with fractional part

```
type Yen is range 0 .. 1_000_000;
type Ruble is range 0 .. 1_000_000;
Mine : Yen := 1;
Yours : Ruble := 1;
Mine := Yours; -- illegal
```

User-Defined Numeric Type Benefits

- Close to **requirements**
  - Types with **explicit** requirements (range, precision, etc.)
  - Best case: Incorrect state **not possible**
- Either implemented/respected or rejected
  - No run-time (bad) suprise
- **Portability** enhanced
  - Reduced hardware dependencies

Summary

- User-defined types and strong typing is **good**
    - Programs written in application's terms
    - Computer in charge of checking constraints
    - Security, reliability requirements have a price
    - Performance **identical**, given **same requirements**
- User definitions from existing types *can* be good
- Right **trade-off** depends on **use-case**
    - More types → more precision → less bugs
    - Storing **both** feet and meters in `Float` has caused bugs
    - More types → more complexity → more bugs
    - A `Green_Round_Object_Altitude` type is probably **never needed**
- Default initialization is **possible**
    - Use **sparingly**

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
   Component1 : Integer;
   Component2 : 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(s) is (are) 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(s) is (are) 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. Run-time 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. Run-time 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. Run-time 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. Run-time error

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

Quiz

```
type Nested_T is record
   Component : 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) 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
   Component : 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) 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

Delta Aggregates

Ada 2022

- A Record can use a *delta aggregate* just like an array

```
type Coordinate_T is record
   X, Y, Z : Float;
end record;
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Prior to Ada 2022, you would copy and then modify

```
declare
   New_Location : Coordinate_T := Location;
begin
   New_Location.Z := 0.0;
   -- OR
   New_Location := (Z => 0.0, others => <>);
end;
```

- Now in Ada 2022 we can just specify the change during the copy

```
New_Location : Coordinate_T := (Location with delta Z => 0.0);
```

Note for record delta aggregates you must use named notation

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

- 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

- 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

```
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

```
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`

Variant Records

Variant Record Types

- *Variant record* can use a **discriminant** to specify alternative lists of components
  - Also called *discriminated record* type
  - Different **objects** may have **different** components
  - All objects **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

Immutable Variant Record

- Discriminant must be set at creation time and cannot be modified

```
2  type Person_Group is (Student, Faculty);
3  type Person (Group : Person_Group) is
4  record
5     --  Components common across all discriminants
6     --  (must appear before variant part)
7     Age : Positive;
8     case Group is --  Variant part of record
9        when Student => -- 1st variant
10          Gpa  : Float range 0.0 .. 4.0;
11       when Faculty => -- 2nd variant
12          Pubs : Positive;
13    end case;
14 end record;
```

- In a variant record, a discriminant can be used to specify the *variant part* (line 8)
  - Similar to case statements (all values must be covered)
  - Components listed will only be visible if choice matches discriminant
  - Component names need to be unique (even across discriminants)
  - Variant part must be end of record (hence only one variant part allowed)
- Discriminant is treated as any other component
  - But is a constant in an immutable variant record

Note that discriminants can be used for other purposes than the variant part

Immutable Variant Record Example

- Each object of Person has three components, but it depends on Group

```
Pat : Person (Student);
Sam : Person := (Faculty, 33, 5);
```

  - Pat has Group, Age, and Gpa
  - Sam has Group, Age, and Pubs
  - Aggregate specifies all components, including the discriminant
- Compiler can detect some problems, but more often clashes are run-time errors

```
procedure Do_Something (Param : in out Person) is
begin
  Param.Age := Param.Age + 1;
  Param.Pubs := Param.Pubs + 1;
end Do_Something;
```

  - Pat.Pubs := 3; would generate a compiler warning because compiler knows Pat is a Student
    - warning: Constraint_Error will be raised at run time
  - Do_Something (Pat); generates a run-time error, because only at runtime is the discriminant for Param known
    - raised CONSTRAINT_ERROR : discriminant check failed
- Pat := Sam; would be a compiler warning because the constraints do not match

Mutable Variant Record

- Type will become *mutable* if its discriminant has a *default value* **and** we instantiate the object without specifying a discriminant

```
type Person_Group is (Student, Faculty);
type Person (Group : Person_Group := Student) is -- default value
record
   Age : Positive;
   case Group is
      when Student =>
         Gpa  : Float range 0.0 .. 4.0;
      when Faculty =>
         Pubs : Positive;
   end case;
end record;
```

- `Pat : Person;` is **mutable**
- `Sam : Person (Faculty);` is **not mutable**
  - Declaring an object with an **explicit** discriminant value (`Faculty`) makes it immutable

Mutable Variant Record Example

- Each object of `Person` has three components, but it depends on `Group`

```
Pat : Person := (Student, 19, 3.9);
Sam : Person (Faculty);
```

- You can only change the discriminant of `Pat`, but only via a whole record assignment, e.g:

```
if Pat.Group = Student then
  Pat := (Faculty, Pat.Age, 1);
else
  Pat := Sam;
end if;
Update (Pat);
```

- But you cannot change the discriminant of `Sam`
  - `Sam := Pat;` will give you a run-time error if `Pat.Group` is not `Faculty`
    - And the compiler will not warn about this!

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. `Variant_Object.N`
C. None: Compilation error
D. None: Run-time error

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. *`Variant_Object.N`*
C. None: Compilation error
D. None: Run-time error

Quiz

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

Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

A. `Variant_Object.F, Variant_Object.Flag`
B. `Variant_Object.F`
C. None: Compilation error
D. None: Run-time error

Quiz

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

Variant_Object : Variant_T (True);
```

Which component does `Variant_Object` contain?

A. `Variant_Object.F, Variant_Object.Flag`
B. `Variant_Object.F`
C. ***None: Compilation error***
D. None: Run-time error

The variant part cannot be followed by a component declaration (`Flag : Character` here)

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 Records

Introduction

Discriminated Record Types

- *Discriminated record* type
  - Different **objects** may have **different** components and/or different sizes
  - All objects **still** share the same type
- Similar to `union` in C
  - But preserves **type checking**
    - Except in the case of an `Unchecked_Union` (seen later)
  - And object size **is related to** discriminant
- Aggregate assignment is allowed
  - Provided constraints are correct

Defining a Discriminated Record

- Record type with a *discriminant*
  - **Discriminant** controls behavior of the record
  - Part of record definition
  - Can be read as any other component
    - But can only be modified by object assignment (sometimes)
- Sample definitions (completions appear later in this module)

```
type Employee_T (Kind : Category_T) is record ...
type Mutable_T (Kind : Category_T := Employee) is record ...
type Vstring (Last : Natural := 0) is record ...
type C_Union_T (View : natural := 0) is record ...
```

Variant Records

What is a Variant Record?

- A *variant record* uses the discriminant to determine which components are currently accessible

```
type Category_T is (Employee, Contractor);
type Employee_T (Kind : Category_T) is record
   Name : String_T;
   DOB  : Date_T;
   case Kind is
      when Employee =>
         Pay_Rate   : Pay_T;
      when Contractor =>
         Hourly_Rate : Contractor_Rate_T;
   end case;
end record;

An_Employee     : Employee_T (Employee);
Some_Contractor : Employee_T (Contractor);
```

- Note that the **case** block must be the last part of the record definition
  - Therefore only one per record
- Variant records are considered the same type
  - So you can have

```
procedure Print (Item : Employee_T);

Print (An_Employee);
Print (Some_Contractor);
```

Immutable Variant Record

- In an *immutable variant record* the discriminant has no default value
    - It is an *indefinite type*, similar to an unconstrained array
        - So you must add a constraint (discriminant) when creating an object
        - But it can be unconstrained when used as a parameter
- For example

```
Pat      : Employee_T (Employee);
Sam      : Employee_T :=
    (Kind        => Contractor,
     Name        => From_String ("Sam"),
     DOB         => "2000/01/01",
     Hourly_Rate => 123.45);
Illegal : Employee_T;  -- indefinite
```

Immutable Variant Record Usage

- Compiler can detect some problems

```
begin
   Pat.Hourly_Rate := 12.3;
end;
```

```
warning: component not present in subtype of
"Employee_T" defined at line 24
```

- But more often clashes are run-time errors

```
32 procedure Print (Item : Employee_T) is
33 begin
34   Print (Item.Pay_Rate);
```

```
raised CONSTRAINT_ERROR : print.adb:34 discriminant
check failed
```

- `Pat := Sam;` would be a compiler warning because the constraints do not match

Mutable Variant Record

- To add flexibility, we can make the type *mutable* by specifying a default value for the discriminant

```
type Mutable_T (Kind : Category_T := Employee) is record
   Name : String_T;
   DOB  : Date_T;
   case Kind is
      when Employee =>
         Pay_Rate   : Pay_T;
      when Contractor =>
         Hourly_Rate : Contractor_Rate_T;
end record;

Pat : Mutable_T;
Sam : Mutable_T (Contractor);
```

- Making the variant mutable creates a definite type
  - An object can be created without a constraint (Pat)
  - Or we can create in immutable object where the discriminant cannot change (Sam)
  - And we can create an array whose component is mutable

Mutable Variant Record Example

- You can only change the discriminant of `Pat`, but only via a whole record assignment, e.g:

```
if Pat.Group = Student then
  Pat := (Faculty, Pat.Age, 1);
else
  Pat := Sam;
end if;
Update (Pat);
```

- But you cannot change the discriminant like a regular component

```
Pat.Kind := Contractor; -- compile error
```

```
error: assignment to discriminant not allowed
```

- And you cannot change the discriminant of `Sam`
  - `Sam := Pat;` will give you a run-time error if `Pat.Kind` is not `Contractor`
    - And the compiler will not warn about this!

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;

Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. `Variant_Object.N`
C. None: Compilation error
D. None: Run-time error

Quiz

```
type Variant_T (Sign : Integer) is record
    case Sign is
    when Integer'First .. -1 =>
        I : Integer;
        B : Boolean;
    when others =>
        N : Natural;
    end case;
end record;
```

```
Variant_Object : Variant_T (1);
```

Which component(s) does `Variant_Object` contain?

A. `Variant_Object.I, Variant_Object.B`
B. *`Variant_Object.N`*
C. None: Compilation error
D. None: Run-time error

Quiz

```
type Coord_T is record
   X, Y : Float;
end record;

type Kind_T is (Circle, Line);
type Shape_T (Kind : Kind_T := Line) is record
   Origin : Coord_T;
   case Kind is
      when Line =>
         End_Point : Coord_T;
      when Circle =>
         End_Point : Coord_T;
   end case;
end record;

A_Circle : Shape_T        :=
  (Circle, (1.0, 2.0), (3.0, 4.0));
A_Line   : Shape_T (Line) :=
   (Circle, (1.0, 2.0), (3.0, 4.0));
```

What happens when you try to build and run this code?

A. Run-time error
B. Compilation error on an object
C. Compilation error on a type
D. No problems

Quiz

```
2   type Coord_T is record
3      X, Y : Float;
4   end record;
5
6   type Kind_T is (Circle, Line);
7   type Shape_T (Kind : Kind_T := Line) is record
8      Origin : Coord_T;
9      case Kind is
10        when Line =>
11           End_Point : Coord_T;
12        when Circle =>
13           End_Point : Coord_T;
14     end case;
15  end record;
16
17  A_Circle : Shape_T        :=
18    (Circle, (1.0, 2.0), (3.0, 4.0));
19  A_Line   : Shape_T (Line) :=
20    (Circle, (1.0, 2.0), (3.0, 4.0));
```

What happens when you try to build and run this code?

A. Run-time error
B. Compilation error on an object
C. ***Compilation error on a type***
D. No problems

- If you fix the compilation error (by changing the name of one of the `End_Point` components), then
  - You would get a warning on line 20 (because `A_Line` is constrained to be a `Line`

    ```
    incorrect value for discriminant "Kind"
    ```

  - If you then ran the executable, you would get an exception

    ```
    CONSTRAINT_ERROR : test.adb:20 discriminant check
    failed
    ```

Discriminant Record Array Size Idiom

Vectors of Varying Lengths

- In Ada, array objects must be fixed length

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

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

Simple Vector of Varying Length

- Not unconstrained - we have to define a maximum length to make it a *definite type*

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

Obj1 : Simple_Vstring := (0, (others => '-'));
Obj2 : Simple_Vstring := (0, (others => '+'));
Obj3 : Simple_Vstring;
```

- Issue - Operations need to consider Last component
  - `Obj1 = Obj2` will be false
  - Can redefine = to be something like

    ```
    if Obj1.Data (1 .. Obj1.Last) = Obj2.Data (1 .. Obj2.Last)
    ```

  - Same thing with concatentation

    ```
    Obj3.Last := Obj1.Last + Obj2.Last;
    Obj3.Data (1 .. Obj3.Last) := Obj1.Data (1 .. Obj1.Last) &
                                  Obj2.Data (1 .. Obj2.Last)
    ```

- Other Issues
  - Every object has same maximum length
  - Last needs to be maintained by program logic

Vector of Varying Length via Discriminated Records

- Discriminant can serve as bound of array component

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

- Mutable objects vs immutable objects
  - With default discriminant value (mutable), objects can be copied even if lengths are different
  - With no default discriminant value (immutable), objects of different lengths cannot be copied (and we can't change the length)

Object Creation

- When a mutable object is created, runtime assumes largest possible value
  - So this example is a problem

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

Good : Vstring (10);
Bad  : Vstring;
```

    - Compiler warning

```
warning: creation of "Vstring" object may raise
Storage_Error
```

    - Run-time error

```
raised STORAGE_ERROR : EXCEPTION_STACK_OVERFLOW
```

- Better implementation

```
subtype Length_T is natural range 0 .. 1_000;
type Vstring (Last : Length_T := 0) is record
   Data   : String (1 .. Last) := (others => ' ');
end record;

Good      : Vstring (10);
Also_Good : Vstring;
```

Simplifying Operations

- With mutable discriminated records, operations are simpler

```
Obj  : Simple_Vstring;
Obj1 : Simple_Vstring := (6, " World");
```

  - Creation

```
function Make (S : String)
  return Vstring is (S'length, S);
Obj2 : Simple_Vstring := Make ("Hello");
```

  - Equality: `Obj1 = Obj2`
    - Data is exactly the correct length
    - if Data or Last is different, equality fails
  - Concatentation

```
Obj := (Obj1.Last + Obj2.Last,
        Obj1.Data & Obj2.Data);
```

Quiz

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

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

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

Quiz

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

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

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

Choices **A** and **B** are mutable: the runtime assumes `Size` can be `Positive'Last`, so component `S` will cause a run-time error. Choice **D** tries to copy a 5-character string into a 6-character string, also generating a run-time error.

Interfacing with C

Passing Records Between Ada and C

- Your Ada code needs to call C that looks like this:

```
struct Struct_T {
   int   Component1;
   char  Component2;
   float Component3;
};

int DoSomething (struct Struct_T);
```

- Ada has mechanisms that will allow you to
  - Call `DoSomething`
  - Build a record that is binary-compatible to `Struct_T`

Building a C-Compatible Record

- To build an Ada record for `Struct_T`, start with a regular record:

```
type Struct_T is record
   Component1 : Interfaces.C.int;
   Component2 : Interfaces.C.char;
   Component3 : Interfaces.C.C_Float;
end record;
```

  - We use types from `Interfaces.C` to map directly to the C types

- But the Ada compiler needs to know that the record layout must match C
  - So we add an aspect to enforce it

```
type Struct_T is record
   Component1 : Interfaces.C.int;
   Component2 : Interfaces.C.char;
   Component3 : Interfaces.C.C_Float;
end record with Convention => C_Pass_By_Copy;
```

Mapping Ada to C Unions

- Discriminant records are similar to C's **union**, but with a limitation
  - Only one part of the record is available at any time
- So, you create the equivalent of this C **union**

```
union Union_T {
   int Component1;
   char Component2;
   float Component3;
};
```

- By using a discriminant record and adding aspect `Unchecked_Union`

```
type C_Union_T (View : natural := 0) is record
   case View is
   when 0 => Component1 : Interfaces.C.int;
   when 1 => Component2 : Interfaces.C.char;
   when 2 => Component3 : Interfaces.C.C_Float;
   when others => null;
   end case;
end record with Convention => C_Pass_By_Copy,
                Unchecked_Union;
```

  - This tells the compiler not to reserve space in the record for the discriminant

Quiz

```
union Union_T {
   struct Record_T component1;
   char            component2[11];
   float           component3;
};
```

```
type C_Union_T (Flag : Natural := 1) is record
    case Sign is
    when 1 =>
        One   : Record_T;
    when 2 =>
        Two   : String(1 .. 11);
    when 3 =>
        Three : Float;
    end case;
end record;
```

```
C_Object : C_Union_T;
```

Which component does `C_Object` contain?

A. `C_Object.One`
B. `C_Object.Two`
C. None: Compilation error
D. None: Run-time error

Quiz

```
union Union_T {
   struct Record_T component1;
   char            component2[11];
   float           component3;
};
```

```
type C_Union_T (Flag : Natural := 1) is record
    case Sign is
    when 1 =>
        One   : Record_T;
    when 2 =>
        Two   : String(1 .. 11);
    when 3 =>
        Three : Float;
    end case;
end record;
```

```
C_Object : C_Union_T;
```

Which component does `C_Object` contain?

A. `C_Object.One`
B. `C_Object.Two`
C. ***None: Compilation error***
D. None: Run-time error

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

Lab

Discriminated Records Lab

- Requirements for a simplistic employee database
  - Create a package to handle varying length strings using variant records
  - 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 Records Lab Solution - Vstring

```
package Vstring is
   Max_String_Length : constant := 1_000;
   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;
   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;
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 Records Lab Solution - Employee (Spec)

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

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

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

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

end Employee;
```

Discriminated Records 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 Records 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
  - Component 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

Array Types

Introduction

What Is an Array?

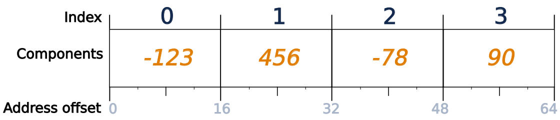
- Definition: collection of components of the same type, stored in contiguous memory, and indexed using a discrete range
- Syntax (simplified):

```
type <typename> is array (Index_Type) of Component_Type;
```

where

- *Index_Type*
  - Discrete range of values to be used to access the array components
- *Component_Type*
  - Type of values stored in the array
  - All components are of this same type and size

```
type Array_T is array (0 .. 3) of Interfaces.Integer_32;
```



Arrays in Ada

- Traditional array concept supported to any dimension

```
declare
   type Hours is digits 6;
   type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   type Schedule is array (Days) of Hours;
   Workdays : Schedule;
begin
   ...
   Workdays (Mon) := 8.5;
```

Array Type Index Constraints

- Must be of an integer or enumeration type
- May be dynamic
- Default to predefined **Integer**
  - Same rules as for-loop parameter default type
- Allowed to be null range
  - Defines an empty array
  - Meaningful when bounds are computed at run-time
- Used to define constrained array types

```
type Schedule is array (Days range Mon .. Fri) of Float;
type Flags_T is array (-10 .. 10) of Boolean;
```

- Or to constrain unconstrained array types

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

Run-Time Index Checking

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

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

Kinds of Array Types

- *Constrained* Array Types
  - Bounds specified by type declaration
  - **All** objects of the type have the same bounds
- *Unconstrained* Array Types
  - Bounds not constrained by type declaration
  - Objects share the type, but not the bounds
  - More flexible

```
type Unconstrained is array (Positive range <>)
  of Integer;

U1 : Unconstrained (1 .. 10);
S1 : String (1 .. 50);
S2 : String (35 .. 95);
```

Constrained Array Types

Constrained Array Type Declarations

Syntax (simplified)

```
type <typename> is array (<index constraint>) of <constrained type>;
```

where

typename - *identifier*

index constraint - *discrete range or type*

constrained type - *type with size known at compile time*

Examples

```
type Integer_Array_T is array (1 .. 3) of Integer;
type Boolean_Array_T is array (Boolean) of Integer;
type Character_Array_T is array (character range 'a' .. 'z') of Boolean;
type Copycat_T is array (Boolean_Array_T'Range) of Integer;
```

Quiz

```
type Array1_T is array (1 .. 8) of Boolean;
type Array2_T is array (0 .. 7) of Boolean;
X1, Y1 : Array1_T;
X2, Y2 : Array2_T;
```

Which statement(s) is (are) legal?

A. `X1 (1) := Y1 (1);`
B. `X1 := Y1;`
C. `X1 (1) := X2 (1);`
D. `X2 := X1;`

Quiz

```
type Array1_T is array (1 .. 8) of Boolean;
type Array2_T is array (0 .. 7) of Boolean;
X1, Y1 : Array1_T;
X2, Y2 : Array2_T;
```

Which statement(s) is (are) legal?

A. *X1 (1) := Y1 (1);*
B. *X1 := Y1;*
C. *X1 (1) := X2 (1);*
D. X2 := X1;

Explanations

A. Legal - components are Boolean
B. Legal - object types match
C. Legal - components are Boolean
D. Although the sizes are the same and the components are the same, the type is different

Unconstrained Array Types

Unconstrained Array Type Declarations

- Do not specify bounds for objects
- Thus different objects of the same type may have different bounds
- Bounds cannot change once set
- Syntax (with simplifications)

```
unconstrained_array_definition ::=
   array (index_subtype_definition
      {, index_subtype_definition})
      of subtype_indication
index_subtype_definition ::= subtype_mark range <>
```

- Examples

```
type Index is range 1 .. Integer'Last;
type Char_Arr is array (Index range <>) of Character;
```

Supplying Index Constraints for Objects

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Schedule is array (Days range <>) of Float;
```

- Bounds set by:
  - Object declaration

    ```
    Weekdays : Schedule(Mon..Fri);
    ```

  - Object (or constant) initialization

    ```
    Weekend : Schedule := (Sat => 4.0, Sun => 0.0);
    -- (Note this is an array aggregate, explained later)
    ```

  - Further type definitions (shown later)
  - Actual parameter to subprogram (shown later)
- Once set, bounds never change

```
Weekdays(Sat) := 0.0; --  Constraint error
Weekend(Mon)  := 0.0; --  Constraint error
```

Bounds Must Satisfy Type Constraints

- Must be somewhere in the range of possible values specified by the type declaration
- `Constraint_Error` otherwise

```
type Index is range 1 .. 100;
type Char_Arr is array (Index range <>) of Character;
...
Wrong : Char_Arr (0 .. 10);  -- run-time error
OK : Char_Arr (50 .. 75);
```

Null Index Range

- When 'Last of the range is smaller than 'First
  - Array is empty - no components
- When using literals, the compiler will allow out-of-range numbers to indicate empty range
  - Provided values are within the index's base type

```
type Index_T is range 1 .. 100;
--  Index_T'Size = 8

type Array_T is array (Index_T range <>) of Integer;

Typical_Empty_Array : Array_T (1 .. 0);
Weird_Empty_Array   : Array_T (123 .. -5);
Illegal_Empty_Array : Array_T (999 .. 0);
```

- When the index type is a single-valued enumerated type, no empty array is possible

"String" Types

- Language-defined unconstrained array types
  - Allow double-quoted literals as well as aggregates
  - Always have a character component type
  - Always one-dimensional
- Language defines various types
  - **String**, with **Character** as component

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

  - **Wide_String**, with **Wide_Character** as component
  - **Wide_Wide_String**, with **Wide_Wide_Character** as component
    - Ada 2005 and later
- Can be defined by applications too

Application-Defined String Types

- Like language-defined string types
  - Always have a character component type
  - Always one-dimensional
- Recall character types are enumeration types with at least one character literal value

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
type Roman_Number is array (Positive range <>)
    of Roman_Digit;
Orwellian : constant Roman_Number := "MCMLXXXIV";
```

Specifying Constraints Via Initial Value

- Lower bound is `Index_subtype'First`
- Upper bound is taken from number of items in value

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>)
    of Character;
...
M : String := "Hello World!";
-- M'First is Positive'First (1)

type Another_String is array (Integer range <>)
    of Character;
...
M : Another_String := "Hello World!";
-- M'First is Integer'First
```

Indefinite Types

- *Indefinite types* do not provide enough information to be instantiated
    - Size
    - Representation
- Unconstrained arrays types are indefinite
    - They do not have a definite `'Size`
- Other indefinite types exist (seen later)

No Indefinite Component Types

- Arrays: consecutive components of the exact **same type**
- Component size must be **defined**
  - No indefinite types
  - No unconstrained types
  - Constrained subtypes allowed

```
type Good is array (1 .. 10) of String (1 .. 20); -- OK
type Bad is array (1 .. 10) of String; -- Illegal
```

Arrays of Arrays

- Allowed (of course!)
  - As long as the "component" array type is constrained
- Indexed using multiple parenthesized values
  - One per array

```
declare
   type Array_of_10 is array (1..10) of Integer;
   type Array_of_Array is array (Boolean) of Array_of_10;
   A : Array_of_Array;
begin
   ...
   A (True)(3) := 42;
```

Quiz

```
type Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are) legal?

A. `AAA : Array_T (0..99);`
B. `BBB : Array_T (1..32);`
C. `CCC : Array_T (17..16);`
D. `DDD : Array_T;`

Quiz

```
type Bit_T is range 0 .. 1;
type Bit_Array_T is array (Positive range <>) of Bit_T;
```

Which declaration(s) is (are) legal?

A. `AAA : Array_T (0..99);`
B. *`BBB : Array_T (1..32);`*
C. *`CCC : Array_T (17..16);`*
D. `DDD : Array_T;`

Explanations

A. `Array_T` index is `Positive` which starts at 1
B. OK, indices are in range
C. OK, indicates a zero-length array
D. Object must be constrained

Attributes

Array Attributes

- Return info about array index bounds

O'Length	number of array components
O'First	value of lower index bound
O'Last	value of upper index bound
O'Range	another way of saying `T'First .. T'Last`

- Meaningfully applied to constrained array types
  - Only constrained array types provide index bounds
  - Returns index info specified by the type (hence all such objects)
- Meaningfully applied to array objects
  - Returns index info for the object
  - Especially useful for objects of unconstrained array types

Attributes' Benefits

- Allow code to be more robust
  - Relationships are explicit
  - Changes are localized
- Optimizer can identify redundant checks

```
declare
   type Int_Arr is array (5 .. 15) of Integer;
   Vector : Int_Arr;
begin
   ...
   for Idx in Vector'Range loop
      Vector (Idx) := Idx * 2;
   end loop;
```

  - Compiler understands `Idx` has to be a valid index for `Vector`, so no run-time checks are necessary

Nth Dimension Array Attributes

- Attribute with **parameter**

```
T'Length (n)
T'First (n)
T'Last (n)
T'Range (n)
```

- `n` is the dimension
  - defaults to 1

```
type Two_Dimensioned is array
   (1 .. 10, 12 .. 50) of T;
TD : Two_Dimensioned;
```

- `TD'First (2) = 12`
- `TD'Last  (2) = 50`
- `TD'Length (2) = 39`
- `TD'First = TD'First (1) = 1`

Quiz

```
subtype Index1_T is Integer range 0 .. 7;
subtype Index2_T is Integer range 1 .. 8;
type Array_T is array (Index1_T, Index2_T) of Integer;
X : Array_T;
```

Which comparison is False?

A. `X'Last (2) = Index2_T'Last`
B. `X'Last (1)*X'Last (2) = X'Length (1)*X'Length (2)`
C. `X'Length (1) = X'Length (2)`
D. `X'Last (1) = 7`

Quiz

```
subtype Index1_T is Integer range 0 .. 7;
subtype Index2_T is Integer range 1 .. 8;
type Array_T is array (Index1_T, Index2_T) of Integer;
X : Array_T;
```

Which comparison is False?

A. `X'Last (2) = Index2_T'Last`
B. ***`X'Last (1)*X'Last (2) = X'Length (1)*X'Length (2)`***
C. `X'Length (1) = X'Length (2)`
D. `X'Last (1) = 7`

Explanations

A. $8 = 8$
B. 7*8 /= 8*8
C. $8 = 8$
D. $7 = 7$

Operations

Object-Level Operations

- Assignment of array objects

```
A := B;
```

- Equality and inequality

```
if A = B then
```

- Conversions
  - Component types must be the same type
  - Index types must be the same or convertible
  - Dimensionality must be the same
  - Bounds must be compatible (not necessarily equal)

```
declare
   type Index1_T is range 1 .. 2;
   type Index2_T is range 101 .. 102;
   type Array1_T is array (Index1_T) of Integer;
   type Array2_T is array (Index2_T) of Integer;
   type Array3_T is array (Boolean) of Integer;

   One   : Array1_T;
   Two   : Array2_T;
   Three : Array3_T;

begin

   One := Array1_T (Two);     -- OK
   Two := Array2_T (Three);   -- Illegal (indices not convertible)
```

Extra Object-Level Operations

- *Only for 1-dimensional arrays!*
- Concatenation

```
type String_Type is array
  (Integer range <>) of Character;
A : constant String_Type := "foo";
B : constant String_Type := "bar";
C : constant String_Type := A & B;
-- C now contains "foobar"
```

- Comparison (for discrete component types)
  - Not for all scalars
- Logical (for `Boolean` component type)
- Slicing
  - Portion of array

Slicing

- Contiguous subsection of an array
- On any **one-dimensional** array type
  - Any component type

```
procedure Test is
  S1 : String (1 .. 9) := "Hi Adam!!";
  S2 : String := "We love    !";
begin
  S2 (9..11) := S1 (4..6);
  Put_Line (S2);
end Test;
```

Result: `We love Ada!`

Example: Slicing with Explicit Indexes

- Imagine a requirement to have a ISO date
  - Year, month, and day with a specific format

```
declare
   Iso_Date : String (1 .. 10) := "2024-03-27";
begin
   Put_Line (Iso_Date);
   Put_Line (Iso_Date (1 .. 4));  --  year
   Put_Line (Iso_Date (6 .. 7));  --  month
   Put_Line (Iso_Date (9 .. 10)); --  day
```

Idiom: Named Subtypes for Indexes

- Subtype name indicates the slice index range
  - Names for constraints, in this case index constraints
- Enhances readability and robustness

```
procedure Test is
  subtype Iso_Index is Positive range 1 .. 10;
  subtype Year is Iso_Index
     range Iso_Index'First .. Iso_Index'First + 3;
  subtype Month is Iso_Index
     range Year'Last + 2 .. Year'Last + 3;
  subtype Day is Iso_Index
     range Month'Last + 2 .. Month'Last + 3;
  Iso_Date : String (Iso_Index) := "2024-03-27";

begin
  Put_Line (Iso_Date (Year));  --  2024
  Put_Line (Iso_Date (Month)); --  03
  Put_Line (Iso_Date (Day));   --  27
```

Dynamic Subtype Constraint Example

- Useful when constraints not known at compile-time
- Example: remove file name extension

```
File_Name
  (File_Name'First
  ..
  Index (File_Name, '.', Direction => Backward));
```

Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

A. `B(1) := A(1,2) or A(4,3);`
B. `B := A(2) and A(4);`
C. `A(1..2)(4) := A(5..6)(8);`
D. `B(3..4) := B(4..5)`

Quiz

```
type Index_T is range 1 .. 10;
type OneD_T is array (Index_T) of Boolean;
type TwoD_T is array (Index_T) of OneD_T;
A : TwoD_T;
B : OneD_T;
```

Which statement(s) is (are) legal?

A. *`B(1) := A(1,2) or A(4,3);`*
B. *`B := A(2) and A(4);`*
C. `A(1..2)(4) := A(5..6)(8);`
D. *`B(3..4) := B(4..5)`*

Explanations

A. All objects are just Boolean values
B. A component of `A` is the same type as `B`
C. Slice must be of outermost array
D. Slicing allowed on single-dimension arrays

Looping Over Array Components

Note on Default Initialization for Array Types

- In Ada, objects are not initialized by default
- To initialize an array, you can initialize each component
  - But if the array type is used in multiple places, it would be better to initialize at the type level
  - No matter how many dimensions, there is only one component type
- Uses aspect **Default_Component_Value**

```
type Vector is array (Positive range <>) of Float
   with Default_Component_Value => 0.0;
```

  - Note that creating a large object of type `Vector` might incur a run-time cost during initialization

Two High-Level For-Loop Kinds

- For arrays and containers
  - Arrays of any type and form
  - Iterable containers
    - Those that define iteration (most do)
    - Not all containers are iterable (e.g., priority queues)!
- For iterator objects
  - Known as "generalized iterators"
  - Language-defined, e.g., most container data structures
- User-defined iterators too
- We focus on the arrays/containers form for now

Array/Container For-Loops

- Work in terms of components within an object
- Syntax hides indexing/iterator controls

```
for name of [reverse] array_or_container_object loop
...
end loop;
```

- Starts with "first" component unless you reverse it
- Loop parameter name is a constant if iterating over a constant, a variable otherwise

Array Component For-Loop Example

- Given an array

```
type T is array (Positive range <>) of Integer;
Primes : T := (2, 3, 5, 7, 11);
```

- Component-based looping would look like

```
for P of Primes loop
   Put_Line (Integer'Image (P));
end loop;
```

- While index-based looping would look like

```
for P in Primes'Range loop
   Put_Line (Integer'Image (Primes (P)));
end loop;
```

Quiz

```
declare
   type Array_T is array (1..5) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in A'First + 1 .. A'Last - 1 loop
      A (I) := I * A'Length;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 10 15 20 1
B. 1 20 15 10 1
C. 0 10 15 20 0
D. 25 20 15 10 5

NB: Without `Default_Component_Value`, init. values are random

Quiz

```
declare
   type Array_T is array (1..5) of Integer
      with Default_Component_Value => 1;
   A : Array_T;
begin
   for I in A'First + 1 .. A'Last - 1 loop
      A (I) := I * A'Length;
   end loop;
   for I of reverse A loop
      Put (I'Image);
   end loop;
end;
```

Which output is correct?

A. 1 10 15 20 1
B. ***1 20 15 10 1***
C. 0 10 15 20 0
D. 25 20 15 10 5

Explanations

A. There is a `reverse`
B. Yes
C. Default value is 1
D. No

NB: Without `Default_Component_Value`, init. values are random

Aggregates

Aggregates

- Literals for composite types
  - Array types
  - Record types
- Two distinct forms
  - Positional
  - Named
- Syntax (simplified):

```
component_expr ::=
  expression -- Defined value
  | <>       -- Default value

array_aggregate ::= (
    {component_expr ,}                        -- Positional
  | {discrete_choice_list => component_expr,}) -- Named
  -- Default "others" indices
  [others => expression]
```

Aggregate "Positional" Form

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

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

Aggregate "Named" Form

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

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

Combined Aggregate Forms Not Allowed

- Some cases lead to ambiguity, therefore never allowed for array types
- Are only allowed for record types (shown in subsequent section)

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

Aggregates Are True Literal Values

- Used any place a value of the type may be used

```
type Schedule is array (Mon .. Fri) of Float;
Work : Schedule;
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0);
...
Work := (8.5, 8.5, 8.5, 8.5, 6.0);
...
if Work = Normal then
...
if Work = (10.0, 10.0, 10.0, 10.0, 0.0) then -- 4-day week
```

Aggregate Consistency Rules

- Must always be complete
    - They are literals, after all
    - Each component must be given a value
    - But defaults are possible (more in a moment)
- Must provide only one value per index position
    - Duplicates are detected at compile-time
- Compiler rejects incomplete or inconsistent aggregates

```
Week := (Sat => False,
         Sun => False,
         Mon .. Fri => True,
         Wed => False);
```

"Others"

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

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

Nested Aggregates

- For arrays of composite component types

```
type Col_T is array (1 .. 3) of Float;
type Matrix_T is array (1 .. 3) of Col_T;
Matrix : Matrix_T := (1 =>  (1.2, 1.3, 1.4),
                      2 =>  (2.5, 2.6, 2.7),
                      3 =>  (3.8, 3.9, 3.0));
```

Defaults Within Array Aggregates

- Specified via the box notation
- Value for component is thus taken as for stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with "named association" form
  - But `others` counts as named form
- Syntax

```
discrete_choice_list => <>
```

- Example

```
type Int_Arr is array (1 .. N) of Integer;
Primes : Int_Arr := (1 => 2, 2 .. N => <>);
```

Named Format Aggregate Rules

- Bounds cannot overlap
  - Index values must be specified once and only once
- All bounds must be static
  - Avoids run-time cost to verify coverage of all index values
  - Except for single choice format

```
type Float_Arr is array (Integer range <>) of Float;
Ages : Float_Arr (1 .. 10) := (1 .. 3 => X, 4 .. 10 => Y);
-- illegal: 3 and 4 appear twice
Overlap : Float_Arr (1 .. 10) := (1 .. 4 => X, 3 .. 10 => Y);
N, M, K, L : Integer;
-- illegal: cannot determine if
-- every index covered at compile time
Not_Static : Float_Arr (1 .. 10) := (M .. N => X, K .. L => Y);
-- This is legal
Values : Float_Arr (1 .. N) := (1 .. N => X);
```

Quiz

```
type Array_T is array (1 .. 5) of Integer;
X : Array_T;
J : Integer := X'First;
```

Which statement is correct?

A. `X := (1, 2, 3, 4 => 4, 5 => 5);`
B. `X := (1..3 => 100, 4..5 => -100, others => -1);`
C. `X := (J => -1, J + 1..X'Last => 1);`
D. `X := (1..3 => 100, 3..5 => 200);`

Quiz

```
type Array_T is array (1 .. 5) of Integer;
X : Array_T;
J : Integer := X'First;
```

Which statement is correct?

A. `X := (1, 2, 3, 4 => 4, 5 => 5);`
B. *`X := (1..3 => 100, 4..5 => -100, others => -1);`*
C. `X := (J => -1, J + 1..X'Last => 1);`
D. `X := (1..3 => 100, 3..5 => 200);`

Explanations

A. Cannot mix positional and named notation
B. Correct - others not needed but is allowed
C. Dynamic values must be the only choice. (This could be fixed by making J a constant.)
D. Overlapping index values (3 appears more than once)

Aggregates in Ada 2022

Ada 2022

- Ada 2022 allows us to use square brackets **"[...]"** in defining aggregates

```
type Array_T is array (positive range <>) of Integer;
```

  - So common aggregates can use either square brackets or parentheses

```
Ada2012 : Array_T := (1, 2, 3);
Ada2022 : Array_T := [1, 2, 3];
```

- But square brackets help in more problematic situations
  - Empty array

```
Ada2012 : Array_T := (1..0 => 0);
Illegal : Array_T := ();
Ada2022 : Array_T := [];
```

  - Single component array

```
Ada2012 : Array_T := (1 => 5);
Illegal : Array_T := (5);
Ada2022 : Array_T := [5];
```

Iterated Component Association

Ada 2022

- With Ada 2022, we can create aggregates with *iterators*
  - Basically, an inline looping mechanism
- Index-based iterator

```
type Array_T is array (positive range <>) of Integer;
Object1 : Array_T(1..5) := (for J in 1 .. 5 => J * 2);
Object2 : Array_T(1..5) := (for J in 2 .. 3 => J,
                            5 => -1,
                            others => 0);
```

  - `Object1` will get initialized to the squares of 1 to 5
  - `Object2` will give the equivalent of `(0, 2, 3, 0, -1)`
- Component-based iterator

```
Object2 := [for Item of Object => Item * 2];
```

  - `Object2` will have each component doubled

More Information on Iterators

Ada 2022

- You can nest iterators for arrays of arrays

```
type Col_T is array (1 .. 3) of Integer;
type Matrix_T is array (1 .. 3) of Col_T;
Matrix : Matrix_T :=
   [for J in 1 .. 3 =>
      [for K in 1 .. 3 => J * 10 + K]];
```

- You can even use multiple iterators for a single dimension array

```
Ada2012 : Array_T(1..5) :=
   [for I in 1 .. 2 => -1,
    for J in 4 ..5 => 1,
    others => 0];
```

- Restrictions
  - You cannot mix index-based iterators and component-based iterators in the same aggregate
  - You still cannot have overlaps or missing values

Delta Aggregates

Ada 2022

```
type Coordinate_T is array (1 .. 3) of Float;
Location : constant Coordinate_T := (1.0, 2.0, 3.0);
```

- Sometimes you want to copy an array with minor modifications
  - Prior to Ada 2022, it would require two steps

```
declare
   New_Location : Coordinate_T := Location;
begin
   New_Location(3) := 0.0;
   -- OR
   New_Location := (3 => 0.0, others => <>);
end;
```

- Ada 2022 introduces a *delta aggregate*
  - Aggregate indicates an object plus the values changed - the *delta*

```
New_Location : Coordinate_T := [Location with delta 3 => 0.0];
```

- Notes
  - You can use square brackets or parentheses
  - Only allowed for single dimension arrays

This works for records as well (see that chapter)

Detour - 'Image for Complex Types

'Image Attribute

Ada 2022

- Previously, we saw the string attribute `'Image` is provided for scalar types
  - e.g. `Integer'Image(10+2)` produces the string **" 12"**
- Starting with Ada 2022, the `Image` attribute can be used for any type

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   type Colors_T is (Red, Yellow, Green);
   type Array_T is array (Colors_T) of Boolean;
   Object : Array_T :=
     (Green  => False,
      Yellow => True,
      Red    => True);
begin
   Put_Line (Object'Image);
end Main;
```

Yields an output of

```
[TRUE, TRUE, FALSE]
```

Overriding the 'Image Attribute

Ada 2022

- We don't always want to rely on the compiler defining how we print a complex object
- We can define it - by using `'Image` and attaching a procedure to the `Put_Image` aspect

```
type Colors_T is (Red, Yellow, Green);
type Array_T is array (Colors_T) of Boolean with
  Put_Image => Array_T_Image;
```

Defining the 'Image Attribute

Ada 2022

- Then we need to declare the procedure

```
procedure Array_T_Image
   (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Value  :        Array_T);
```

  - Which uses the `Ada.Strings.Text_Buffers.Root_Buffer_Type` as an output buffer
  - (No need to go into detail here other than knowing you do `Output.Put` to add to the buffer)

- And then we define it

```
procedure Array_T_Image
   (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Value  :        Array_T) is
begin
   for Color in Value'Range loop
      Output.Put (Color'Image & "=>" & Value (Color)'Image & ASCII.LF);
   end loop;
end Array_T_Image;
```

Using the 'Image Attribute

Ada 2022

- Now, when we call Image we get our "pretty-print" version

```
with Ada.Text_IO; use Ada.Text_IO;
with Types; use Types;
procedure Main is
   Object : Array_T := (Green  => False,
                        Yellow => True,
                        Red    => True);
begin
   Put_Line (Object'Image);
end Main;
```

  - Generating the following output

```
RED=>TRUE
YELLOW=>TRUE
GREEN=>FALSE
```

- Note this redefinition can be used on any type, even the scalars that have always had the attribute

Anonymous Array Types

Anonymous Array Types

- Array objects need not be of a named type
  `A : array (1 .. 3) of B;`
- Without a type name, no object-level operations
  - Cannot be checked for type compatibility
  - Operations on components are still ok if compatible

```
declare
-- These are not same type!
  A, B : array (Foo) of Bar;
begin
  A := B;  -- illegal
  B := A;  -- illegal
  -- legal assignment of value
  A(J) := B(K);
end;
```

Lab

Array Lab

- Requirements
  - Create an array type whose index is days of the week and each component is a number
  - Create two objects of the array type, one of which is constant
  - Perform the following operations
    - Copy the constant object to the non-constant object
    - Print the contents of the non-constant object
    - Use an array aggregate to initialize the non-constant object
    - For each component of the array, print the array index and the value
    - Move part ("source") of the non-constant object to another part ("destination"), and then clear the source location
    - Print the contents of the non-constant object
- Hints
  - When you want to combine multiple strings (which are arrays!) use the concatenation operator (**&**)
  - Slices are how you access part of an array
  - Use aggregates (either named or positional) to initialize data

Arrays of Arrays

- Requirements
  - For each day of the week, you need an array of three strings containing names of workers for that day
  - Two sets of workers: weekend and weekday, but the store is closed on Wednesday (no workers)
  - Initialize the array and then print it hierarchically

Array Lab Solution - Declarations

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

   type Days_Of_Week_T is
      (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   type Unconstrained_Array_T is
      array (Days_Of_Week_T range <>) of Natural;

   Const_Arr : constant Unconstrained_Array_T := (1, 2, 3, 4
   Array_Var : Unconstrained_Array_T (Days_Of_Week_T);

   type Name_T is array (1 .. 6) of Character;
   type Names_T is array (1 .. 3) of Name_T;
   Weekly_Staff : array (Days_Of_Week_T) of Names_T;
```

Array Lab Solution - Implementation

```
begin
   Array_Var := Const_Arr;
   for Item of Array_Var loop
      Put_Line (Item'Image);
   end loop;
   New_Line;

   Array_Var :=
     (Mon => 111, Tue => 222, Wed => 333, Thu => 444, Fri => 555, Sat => 666,
      Sun => 777);
   for Index in Array_Var'Range loop
      Put_Line (Index'Image & " => " & Array_Var (Index)'Image);
   end loop;
   New_Line;

   Array_Var (Mon .. Wed) := Const_Arr (Wed .. Fri);
   Array_Var (Wed .. Fri) := (others => Natural'First);
   for Item of Array_Var loop
      Put_Line (Item'Image);
   end loop;
   New_Line;

   Weekly_Staff := (Mon | Tue | Thu | Fri => ("Fred  ", "Barney", "Wilma "),
                    Wed    => ("closed", "closed", "closed"),
                    others => ("Pinky ", "Inky  ", "Blinky"));

   for Day in Weekly_Staff'Range loop
      Put_Line (Day'Image);
      for Staff of Weekly_Staff(Day) loop
         Put_Line ("  " & String (Staff));
      end loop;
   end loop;
end Main;
```

Summary

Final Notes on Type **String**

- Any single-dimensioned array of some character type is a *string type*
  - Language defines types **String**, **Wide_String**, etc.
- Just another array type: no null termination
- Language-defined support defined in Appendix A
  - **Ada.Strings.***
  - Fixed-length, bounded-length, and unbounded-length
  - Searches for pattern strings and for characters in program-specified sets
  - Transformation (replacing, inserting, overwriting, and deleting of substrings)
  - Translation (via a character-to-character mapping)

Summary

- Any dimensionality directly supported
- Component types can be any (constrained) type
- Index types can be any discrete type
  - Integer types
  - Enumeration types
- Constrained array types specify bounds for all objects
- Unconstrained array types leave bounds to the objects
  - Thus differently-sized objects of the same type
- Default initialization for large arrays may be expensive!
- Anonymously-typed array objects used in examples for brevity but that doesn't mean you should in real programs

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

Reminder: What is a Type?

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

```
package Types is
   type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;
   procedure Increment_With_Truncation (Val : in out Integer_T);
   procedure Increment_With_Rounding (Val : in out Integer_T);
end Types;
```

Simple Derivation

Simple Type Derivation

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

```
type Natural_T is new Integer_T range 0 .. Integer_T'Last;
```

- `Natural_T` inherits from:
  - The data **representation** of the parent
    - Integer based, 64 bits
  - The **primitives** of the parent
    - `Increment_With_Truncation` and `Increment_With_Rounding`
- The types are not the same

```
I_Obj : Integer_T := 0;
N_Obj : Natural_T := 0;
```

  - `I_Obj := N_Obj;` → generates a compile error

    expected type "Integer_T" defined at line 2
  - But a child can be converted to the parent
    - `I_Obj := Integer_T (N_Obj);`

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 Positive_T is new Natural_T range 1 .. Natural_T'Last;
```

- Unconstrained types can be constrained

```
type Arr_T is array (Integer range <>) of Integer;
type Ten_Elem_Arr_T is new Arr_T (1 .. 10);
type Rec_T (Size : Integer) is record
   Elem : Arr_T (1 .. Size);
end record;
type Ten_Elem_Rec_T is new Rec_T (10);
```

Primitives

Primitive Operations

- Primitive Operations are those subprograms associated with a type

```
type Integer_T is range -(2**63) .. 2**63-1 with Size => 64;
procedure Increment_With_Truncation (Val : in out Integer_T);
procedure Increment_With_Rounding (Val : in out Integer_T);
```

- Most types have some primitive operations defined by the language
  - e.g. equality operators for most types, numeric operators for integers and floats
- A primitive operation on the parent can receive an object of a child type with no conversion

```
declare
   N_Obj : Natural_T := 1234;
begin
   Increment_With_Truncation (N_Obj);
end;
```

General Rule for Defining a Primitive

- Primitives are subprograms
- Subprogram `S` is a primitive of type `T` if and only if:
  - `S` is declared in the scope of `T`
  - `S` uses type `T`
    - As a parameter
    - As its return type (for a `function`)
  - `S` is above *freeze-point* (see next section)
- Standard practice
  - Primitives should 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;
```

Primitive of Multiple Types

A subprogram can be a primitive of several types

```
package P is
   type Distance_T is range 0 .. 9999;
   type Percentage_T is digits 2 range 0.0 .. 1.0;
   type Units_T is (Meters, Feet, Furlongs);

   procedure Convert (Value  : in out Distance_T;
                      Source :        Units_T;
                      Result :        Units_T);
   procedure Shrink (Value   : in out Distance_T;
                     Percent :        Percentage_T);

end P;
```

- Convert and Shrink are primitives for Distance_T
- Convert is also a primitive of Units_T
- Shrink is also a primitive of Percentage_T

Creating Primitives for Children

- Just because we can inherit a primitive from out parent doesn't mean we want to
- We can create a new primitive (with the same name as the parent) for the child
  - Very similar to overloaded subprograms
  - But added benefit of visibility to grandchildren
- We can also remove a primitive (see next slide)

```
type Integer_T is range -(2**63) .. 2**63-1;
procedure Increment_With_Truncation (Val : in out Integer_T);
procedure Increment_With_Rounding (Val : in out Integer_T);

type Child_T is new Integer_T range -1000 .. 1000;
procedure Increment_With_Truncation (Val : in out Child_T);

type Grandchild_T is new Child_T range -100 .. 100;
procedure Increment_With_Rounding (Val : in out Grandchild_T);
```

Overriding Indications

- **Optional** indications
- Checked by compiler

```
type Child_T is new Integer_T range -1000 .. 1000;
procedure Increment_With_Truncation
   (Val : in out Child_T);
procedure Just_For_Child
   (Val : in out Child_T);
```

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

```
overriding procedure Increment_With_Truncation
   (Val : in out Child_T);
```

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

```
not overriding procedure Just_For_Child
   (Val : in out Child_T);
```

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

```
overriding procedure Just_For_Child
   (Val : in out Grandchild_T) is abstract;
```

- Using `overriding` or `not overriding` incorrectly will generate a compile error

Quiz

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

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

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

Quiz

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

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

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

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

Freeze Point

What is the "Freeze Point"?

- Ada doesn't explicitly identify the end of the "scope" of a type
  - The compiler needs to know it for determining primitive operations
  - Also needed for other situations (described elsewhere)
- This end is the implicit **freeze point** occurring whenever:
  - A **variable** of the type is **declared**
  - The type is **derived**
  - The **end of the scope** is reached
- Subprograms past this "freeze point" are not primitive operations

```
type Parent is Integer;
procedure Prim (V : Parent);

type Child is new Parent;

-- Parent has been derived, so it is frozen.
-- Prim2 is not a primitive
procedure Prim2 (V : Parent);

V : Child;

-- Child used in an object declaration, so it is frozen
-- Prim3 is not a primitive
procedure Prim3 (V : Child);
```

Debugging Type Freeze

- Freeze → Type **completely** defined
- Compiler does **need** to determine the freeze point
  - To instantiate, derive, get info on the type (`'Size`)...
  - Freeze rules are a guide to place it
  - Actual choice is more technical
    - May contradict the standard
- `-gnatDG` to get **expanded** source
  - **Pseudo-Ada** debug information

pkg.ads

```
type Up_To_Eleven is range 0 .. 11;
```

<obj>/pkg.ads.dg

```
type example__up_to_eleven_t is range 0 .. 11;             -- type declaration
[type example__Tup_to_eleven_tB is new short_short_integer] -- representation
freeze example__Tup_to_eleven_tB []                         -- freeze representation
freeze example__up_to_eleven_t []                           -- freeze representation
```

Quiz

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

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

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

type Grandchild is new Child range 3 .. 98;

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

Which are `Parent`'s primitives?

A. `Proc_A`
B. `Proc_B`
C. `Proc_C`
D. No primitives of `Parent`

Quiz

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

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

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

type Grandchild is new Child range 3 .. 98;

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

Which are Parent's primitives?

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

Explanations

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

Summary

Summary

- *Primitive* of a type
  - Subprogram above **freeze-point** that takes or returns 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

Expressions

Introduction

Advanced Expressions

- Different categories of expressions above simple assignment and conditional statements
  - Constraining types to sub-ranges to increase readability and flexibility
    - Allows for simple membership checks of values
  - Embedded conditional assignments
    - Equivalent to C's `A ? B : C` and even more elaborate

Membership Tests

"Membership" Operation

- Syntax

```
simple_expression [not] in membership_choice_list
membership_choice_list ::= membership_choice
                           { | membership_choice}
membership_choice ::= expression | range | subtype_mark
```

- Acts like a boolean function
- Usable anywhere a boolean value is allowed

```
X : Integer := ...
B : Boolean := X in 0..5;
C : Boolean := X not in 0..5; -- also "not (X in 0..5)"
```

Testing Constraints Via Membership

```
type Calendar_Days  is
    (Mon, Tues, Wed, Thur, Fri, Sat, Sun);
subtype Weekdays is Calendar_Days range Mon .. Fri;
Day : Calendar_Days := Today;
...
if Day in Mon .. Fri then ...
if Day in Weekdays then ... -- same as above
```

Testing Non-Contiguous Membership

- We use `in` to indicate membership in a range of values

```
if Color in Red .. Green then
if Index in List'Range then
```

- But what if the values are not contiguous?
  - We could use a Boolean conjunction

```
if Index = 1 or Index = 3 or Index = 5 then
```

  - Or we could simplify it by specifying a collection (or set)

```
if Index in 1 | 3 | 5 then
```

    - | is used to separate members
    - So `1 | 3 | 5` is the set for which we are verifying membership

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Weekdays_T is Days_T range Mon .. Fri;
Today : Days_T;
```

Which condition(s) is (are) legal?

A. `if Today = Mon or Wed or Fri then`
B. `if Today in Days_T then`
C. `if Today not in Weekdays_T then`
D. `if Today in Tue | Thu then`

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
subtype Weekdays_T is Days_T range Mon .. Fri;
Today : Days_T;
```

Which condition(s) is (are) legal?

A. `if Today = Mon or Wed or Fri then`
B. *`if Today in Days_T then`*
C. *`if Today not in Weekdays_T then`*
D. *`if Today in Tue | Thu then`*

Explanations

A. `Wed` and `Fri` are not Boolean expressions - need to compare each of them to `Today`
B. Legal - should always return `True`
C. Legal - returns `True` if `Today` is `Sat` or `Sun`
D. Legal - returns `True` if `Today` is `Tue` or `Thu`

Qualified Names

Qualification

- Explicitly indicates the subtype of the value
- Syntax

```
qualified_expression ::= subtype_mark'(expression) |
                         subtype_mark'aggregate
```

- Similar to conversion syntax
  - Mnemonic - "qualification uses quote"
- Various uses shown in course
  - Testing constraints
  - Removing ambiguity of overloading
  - Enhancing readability via explicitness

Testing Constraints Via Qualification

- Asserts value is compatible with subtype
  - Raises exception `Constraint_Error` if not true

```
subtype Weekdays is Days range Mon .. Fri;
This_Day : Days;
...
case Weekdays'(This_Day) is -- run-time error if out of range
  when Mon =>
    Arrive_Late;
    Leave_Early;
  when Tue .. Thur =>
    Arrive_Early;
    Leave_Late;
  when Fri =>
    Arrive_Early;
    Leave_Early;
end case; -- no 'others' because all subtype values covered
```

Conditional Expressions

Conditional Expressions

- Ultimate value depends on a controlling condition
- Allowed wherever an expression is allowed
  - Assignment RHS, formal parameters, aggregates, etc.
- Similar intent as in other languages
  - Java, C/C++ ternary operation **A ? B : C**
  - Python conditional expressions
  - etc.
- Two forms:
  - *If expressions*
  - *Case expressions*

If Expressions

- Syntax looks like an *if statement* without **end if**

```
if_expression ::=
   (if condition then dependent_expression
   {elsif condition then dependent_expression}
   [else dependent_expression])
condition ::= boolean_expression
```

  - The conditions are always Boolean values

```
(if Today > Wednesday then 1 else 0)
```

Result Must Be Compatible with Context

- The **dependent_expression** parts, specifically

```
X : Integer :=
    (if Day_Of_Week (Clock) > Wednesday then 1 else 0);
```

"If Expression" Example

```
declare
  Remaining : Natural := 5;  -- arbitrary
begin
  while Remaining > 0 loop
    Put_Line ("Warning! Self-destruct in" &
      Remaining'Image &
      (if Remaining = 1 then " second" else " seconds"));
    delay 1.0;
    Remaining := Remaining - 1;
  end loop;
  Put_Line ("Boom! (goodbye Nostromo)");
```

Boolean "If Expressions"

- Return a value of either True or False
  - `(if P then Q)` - assuming **P** and **Q** are **Boolean**
  - "If P is True then the result of the *if expression* is the value of Q"
- But what is the overall result if all conditions are False?
- Answer: the default result value is True
  - Why?
    - Consistency with mathematical proving

The "else" Part When Result Is Boolean

- Redundant because the default result is True

```
(if P then Q else True)
```

- So for convenience and elegance it can be omitted

```
Acceptable : Boolean := (if P1 > 0 then P2 > 0 else True);
Acceptable : Boolean := (if P1 > 0 then P2 > 0);
```

- Use `else` if you need to return False at the end

Rationale for Parentheses Requirement

- Prevents ambiguity regarding any enclosing expression
- Problem:

```
X : Integer := if condition then A else B + 1;
```

- Does that mean
  - If condition, then **X := A + 1**, else **X := B + 1 OR**
  - If condition, then **X := A**, else **X := B + 1**
- But not required if parentheses already present
  - Because enclosing construct includes them

```
Subprogram_Call (if A then B else C);
```

When to Use If Expressions

- When you need computation to be done prior to sequence of statements
  - Allows constants that would otherwise have to be variables
- When an enclosing function would be either heavy or redundant with enclosing context
  - You'd already have written a function if you'd wanted one
- Preconditions and postconditions
  - All the above reasons
  - Puts meaning close to use rather than in package body
- Static named numbers
  - Can be much cleaner than using Boolean'Pos (Condition)

"If Expression" Example for Constants

- Starting from

```
End_of_Month : array (Months) of Days
  := (Sep | Apr | Jun | Nov => 30,
     Feb => 28,
     others => 31);
begin
  if Leap (Today.Year) then -- adjust for leap year
    End_of_Month (Feb) := 29;
  end if;
  if Today.Day = End_of_Month (Today.Month) then
...
```

- Using *if expression* to call `Leap (Year)` as needed

```
End_Of_Month : constant array (Months) of Days
  := (Sep | Apr | Jun | Nov => 30,
      Feb => (if Leap (Today.Year)
              then 29 else 28),
      others => 31);
begin
  if Today.Day /= End_of_Month (Today.Month) then
...
```

Case Expressions

- Syntax similar to *case statements*
  - Lighter: no closing **end case**
  - Commas between choices
- Same general rules as *if expressions*
  - Parentheses required unless already present
  - Type of "result" must match context
- Advantage over *if expressions* is completeness checked by compiler
- Same as with `case` statements (unless `others` is used)

```
-- compile error if not all days covered
Hours : constant Integer :=
   (case Day_of_Week is
    when Mon .. Thurs => 9,
    when Fri          => 4,
    when Sat | Sun    => 0);
```

"Case Expression" Example

```
Leap : constant Boolean :=
   (Today.Year mod 4 = 0 and Today.Year mod 100 /= 0)
   or else
   (Today.Year mod 400 = 0);
End_Of_Month : array (Months) of Days;
...
-- initialize array
for M in Months loop
  End_Of_Month (M) :=
     (case M is
      when Sep | Apr | Jun | Nov => 30,
      when Feb => (if Leap then 29 else 28),
      when others => 31);
end loop;
```

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
```

Which statement(s) is (are) legal?

A. `F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);`
B. `F := Sqrt (if X < 0.0 then -1.0 * X else X);`
C. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`
D. `B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`

Quiz

```
function Sqrt (X : Float) return Float;
F : Float;
B : Boolean;
```

Which statement(s) is (are) legal?

A. `F := if X < 0.0 then Sqrt (-1.0 * X) else Sqrt (X);`
B. ***`F := Sqrt (if X < 0.0 then -1.0 * X else X);`***
C. ***`B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0 else True);`***
D. ***`B := (if X < 0.0 then Sqrt (-1.0 * X) < 10.0);`***

Explanations

A. Missing parentheses around expression
B. Legal - Expression is already enclosed in parentheses so you don't need to add more
C. Legal - **`else`** `True` not needed but is allowed
D. Legal - `B` will be `True` if X >= 0.0

Quantified Expressions

Introduction

- 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

Semantics Are As If You Wrote This Code

```
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

- 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

```
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

- 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

- "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

```
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

- 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

- "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

- 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

- 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...

- 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

- 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

- 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");*

```
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 components are arrays of three components. Which expression would one use to determine if at least one of A's components 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 components are arrays of three components. Which expression would one use to determine if at least one of A's components 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 component has two consecutive increasing values
B. Will be `True` if every component is sorted
C. Correct
D. Will be `True` if every component has two consecutive increasing values

Lab

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!

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);
```

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

- Conditional expressions are allowed wherever expressions are allowed, but beware over-use
  - Especially useful when a constant is intended
  - Especially useful when a static expression is required
- 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 components
    - 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(s) is (are) 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(s) is (are) 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

- *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

```
--  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

- 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

- **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.
```
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

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

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

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

D.
```ada
type Priv is record
  Component : 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
  Component : 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
     Component : Integer;
  end record;
  type L2_T is record
     Component : Integer;
  end record;
  type P1_T is limited record
     Component : L1_T;
  end record;
  type P2_T is record
     Component : L2_T;
  end record;
end P;
```

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
     Component : Integer;
  end record;
  type L2_T is record
     Component : Integer;
  end record;
  type P1_T is limited record
     Component : L1_T;
  end record;
  type P2_T is record
     Component : L2_T;
  end record;
end P;
```

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

Full definition of `P1_T` adds restrictions, which is not allowed. `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)

```
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

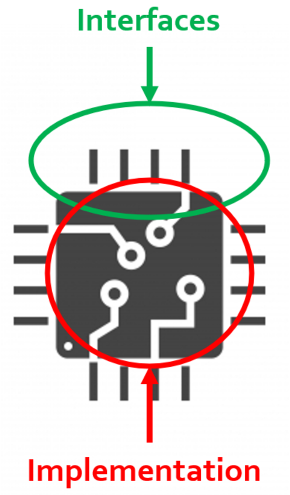
Introduction

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

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 : Bounded_Stacks.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(s) is (are) legal?

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

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

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

```
    end record;
```

Quiz

```
package P is
   type Private_T is private;

   type Record_T is record
```

Which component(s) is (are) legal?

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

B. `Component_B : Private_T := null;`

C. `Component_C : Private_T := 0;`

D. *`Component_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 and Recompilation

- Users can compile their code before the package body is compiled or even written
- 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(s) is (are) 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(s) is (are) 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

- Reminder: view is the *interface* you have on the type

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

- **Designer** of package has **Full** view
  - **Once** completion is reached
  - All operations based upon **full definition** of type

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.Instance := 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 components is annoying

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

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 component 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 component
  - 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_Component_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_Component_T;
   function Image (Item : Map_Component_T) return String;
   function Image (Flag : Map_T) return String;
private
   type Map_Component_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_Component_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)

```
function Find (Map : Map_T;
               Key : Key_T)
               return Integer is
begin
   for I in 1 .. Map.Length loop
      if Map.Values (I).Key = Key then
         return I;
      end if;
   end loop;
   return -1;
end Find;

procedure Add (Map         : in out Map_T;
               Key         :        Key_T;
               Description :        Colors.Color_Set_T;
               Success     :    out Boolean) is
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index not in Map.Values'Range then
      declare
         New_Item : constant Map_Component_T :=
           (Key         => Key,
            Description => Description);
      begin
         Map.Length              := Map.Length + 1;
         Map.Values (Map.Length) := New_Item;
         Success                 := True;
      end;
   end if;
end Add;

procedure Remove (Map     : in out Map_T;
                  Key     :        Key_T;
                  Success :    out Boolean) is
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index in Map.Values'Range then
      Map.Values (Index .. Map.Length - 1) :=
        Map.Values (Index + 1 .. Map.Length);
      Success                          := True;
   end if;
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
   Index : constant Integer := Find (Map, Key);
begin
   Success := False;
   if Index in Map.Values'Range then
      Map.Values (Index).Description := Description;
      Success                        := True;
   end if;
end Modify;

function Exists (Map : Map_T;
                 Key : Key_T)
                 return Boolean is
   (Find (Map, Key) in Map.Values'Range);

function Get (Map : Map_T;
              Key : Key_T)
              return Map_Component_T is
   Index   : constant Integer := Find (Map, Key);
   Ret_Val : Map_Component_T;
begin
   if Index in Map.Values'Range then
      Ret_Val := Map.Values (Index);
   end if;
   return Ret_Val;
end Get;

function Image (Item : Map_Component_T) return String is
  (Item.Key'Image & " => " & 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 I in 1 .. Flag.Length loop
      declare
         Item : constant Map_Component_T := Flag.Values (I);
         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 In Depth

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 - General vs Pool-Specific

General Access Types

- Point to any object of designated type
- Useful for creating aliases to existing objects
- Point to existing object via `'Access` **or** created by `new`
- No automatic memory management

Pool-Specific Access Types

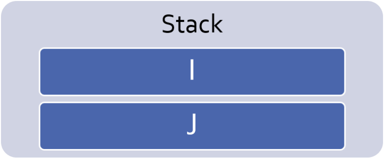
- Tightly coupled to dynamically allocated objects
- Used with Ada's controlled memory management (pools)
- Can only point to object created by `new`
- Memory management tied to specific storage pool

Access Types Can Be Dangerous

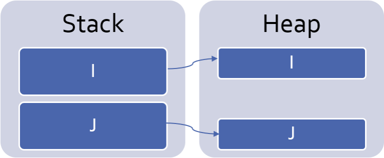
- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot 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
    - Details elsewhere

```
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;
   --  Necessary to avoid corruption
   --  Watch out for any of G's copies!
   G := null;
end P1;

procedure P2 is
begin
   G.all := 5;
end P2;
```

Aliased Parameters

- To ensure a subprogram parameter always has a valid memory address, define it as **aliased**
  - Ensures 'Access and 'Address are valid for the parameter

```
procedure Example (Param : aliased Integer);

Object1 : aliased Integer;
Object2 : Integer;

-- This is OK
Example (Object1);

-- Compile error: Object2 could be optimized away
-- or stored in a register
Example (Object2);

-- Compile error: No address available for parameter
Example (123);
```

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(s) is (are) 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(s) is (are) 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)

- Issues with nesting

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

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

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

Dynamic Accessibility Checks

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

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

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

Getting Around Accessibility Checks

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

```
type Acc is access all Integer;
G : Acc;
procedure P is
   V : aliased Integer;
begin
   G := V'Unchecked_Access;
   ...
   Do_Something (G.all);
   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_Access  : Global_Access_T;
Global_Object  : aliased Integer;
procedure Proc_Access is
   type Local_Access_T is access all Integer;
   Local_Access  : Local_Access_T;
   Local_Object  : aliased Integer;
begin
```

Which assignment(s) is (are) legal?

A. `Global_Access := Global_Object'Access;`
B. `Global_Access := Local_Object'Access;`
C. `Local_Access  := Global_Object'Access;`
D. `Local_Access  := Local_Object'Access;`

Quiz

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

Which assignment(s) is (are) legal?

A. *`Global_Access := Global_Object'Access;`*
B. `Global_Access := Local_Object'Access;`
C. *`Local_Access  := Global_Object'Access;`*
D. *`Local_Access  := Local_Object'Access;`*

Explanations

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

Memory Corruption

Common Memory Problems (1/3)

- Uninitialized pointers

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

- Double deallocation

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

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

Common Memory Problems (2/3)

- Accessing deallocated memory

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

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

Common Memory Problems (3/3)

- Memory leaks

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

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

How to Fix Memory Problems?

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

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
```

Memory Management

Simple Linked List

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

Incomplete Types

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

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

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

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

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

Linked List in Ada

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

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

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

Simplistic Linked List

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

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

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

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

begin

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

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

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

end Simple;
```

Memory Debugging

GNAT.Debug_Pools

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

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

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

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

GNAT.Debug_Pools Spec (Partial)

```
package GNAT.Debug_Pools is

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

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

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

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

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

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

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

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

Displaying Debug Information

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

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

  - Dump info after each `new`

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

  - Dump info after free

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

Execution Results

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

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

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

Memory Control

System.Storage_Pools

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

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

System.Storage_Pools Spec (Partial)

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

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

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

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

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

System.Storage_Pools Explanations

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

System.Storage_Pools Example (Partial)

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

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

function Find_Free_Block (Length : Storage_Count) return Index_T;

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

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

Advanced Access Type Safety

Elaboration-Only Dynamic Allocation

- Common in critical contexts
- Rationale:
  1. We (might) need dynamically allocated date
     - e.g. loading configuration data of unknown size
  2. Deallocations can cause leaks, corruption
     - → **Disallow** them entirely
  3. A dynamically allocated object will need deallocation
     - → Unless it never goes out of **scope**
- → Allow only allocation onto globals

Tip

And restrict allocations to program elaboration

Prevent Heap Deallocations

- `Ada.Unchecked_Deallocation` cannot be used anymore
- No heap deallocation is possible
  - The total number of allocations should be bounded
  - e.g. elaboration-only allocations

```
pragma Restrictions
    (No_Dependence => Unchecked_Deallocation);
```

Constant Access at Library Level

```
type Acc is access T;
procedure Free is new Ada.Unchecked_Deallocation (T, Acc);

A : constant Acc := new T;
```

- `A` is **constant**
    - Cannot be deallocated

Constant Access as Discriminant

```
type R (A : access T) is limited record
```

- `A` is **`constant`**
  - Cannot be deallocated
- `R` is **`limited`**
  - Cannot be copied

Idiom: Access to Subtype

Tip

subtype improves access-related code safety

- Subtype constraints still apply through the access type

```
type Values_T is array (Positive range <>) of Integer;
subtype Two_Values_T is Values_T (1 .. 2);
type Two_Values_A is access all Two_Values_T;

function Get return Values_T is (1 => 10);

-- O : aliased Two_Values_T := Get;
-- Runtime FAIL: Constraint check
O : aliased Values_T := Get; -- Single value, bounds are 1 .. 
-- P : Two_Values_A := O'Access;
-- Compile-time FAIL: Bounds must statically match
```

Lab

Access Types In Depth Lab

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

Extra Credit

- Complete as many of these as you have time for

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

   - Should still be able to print memory status

Lab Solution - Database

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

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

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

Lab Solution - Database_List (Spec)

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

Lab Solution - Database_List (Helper Objects)

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

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

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

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

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

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

Lab Solution - Database_List (Insert/Delete)

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

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

Lab Solution - Main

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

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

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

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

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

Lab Solution - Simple_IO (Spec)

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

Lab Solution - Simple_IO (Body)

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

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

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

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

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

Lab Solution - Memory_Mgmt (Debug Pools)

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

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

Lab Solution - Memory_Mgmt (Storage Pools Spec)

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

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

   procedure Print_Info;

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

   Storage_Pool : Storage_Pool_T;

end Memory_Mgmt;
```

Lab Solution - Memory_Mgmt (Storage Pools 1/2)

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

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

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

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

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

Lab Solution - Memory_Mgmt (Storage Pools 2/2)

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

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

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

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

end Memory_Mgmt;
```

Summary

Summary

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

Controlled Types

Introduction

Constructor / Destructor

- Possible to specify behavior of object initialization, finalization, and assignment
  - Based on type definition
  - Type must derive from **Controlled** or **Limited_Controlled** in package **Ada.Finalization**
- This derived type is called a *controlled type*
  - User may override any or all subprograms in **Ada.Finalization**
  - Default implementation is a null body

Ada.Finalization

Package Spec

```
package Ada.Finalization is

  type Controlled is abstract tagged private;
  procedure Initialize (Object : in out Controlled)
     is null;
  procedure Adjust    (Object : in out Controlled)
     is null;
  procedure Finalize  (Object : in out Controlled)
     is null;

  type Limited_Controlled is abstract tagged limited private;
  procedure Initialize (Object : in out Limited_Controlled)
     is null;
  procedure Finalize  (Object : in out Limited_Controlled)
     is null;

private
   -- implementation defined
end Ada.Finalization;
```

Uses

- Prevent "resource leak"
  - Logic centralized in service rather than distributed across clients
- Examples: heap reclamation, "mutex" unlocking
- User-defined assignment

Initialization

- Subprogram **Initialize** invoked after object created
  - Either by object declaration or allocator
  - Only if no explicit initialization expression
- Often default initialization expressions on record components are sufficient
  - No need for an explicit call to **Initialize**
- Similar to C++ constructor

Finalization

- Subprogram **Finalize** invoked just before object is destroyed
  - Leaving the scope of a declared object
  - Unchecked deallocation of an allocated object
- Similar to C++ destructor

Assignment

- Subprogram **Adjust** invoked as part of an assignment operation
- Assignment statement **Target := Source;** is basically:
  - **Finalize (Target)**
  - Copy Source to Target
  - **Adjust (Target)**
  - *Actual rules are more complicated, e.g. to allow cases where Target and Source are the same object*
- Typical situations where objects are access values
  - **Finalize** does unchecked deallocation or decrements a reference count
  - The copy step copies the access value
  - **Adjust** either clones a "deep copy" of the referenced object or increments a reference count

Example

Unbounded String Via Access Type

- Type contains a pointer to a string type
- We want the provider to allocate and free memory "safely"
  - No sharing
  - **Adjust** allocates referenced String
  - **Finalize** frees the referenced String
  - Assignment deallocates target string and assigns copy of source string to target string

Unbounded String Usage

```
with Unbounded_String_Pkg; use Unbounded_String_Pkg;
procedure Test is
   U1 : Ustring_T;
begin
   U1 := To_Ustring_T ("Hello");
   declare
      U2 : Ustring_T;
   begin
      U2 := To_Ustring_T ("Goodbye");
      U1 := U2; -- Reclaims U1 memory
   end; -- Reclaims U2 memory
end Test; -- Reclaims U1 memory
```

Unbounded String Definition

```
with Ada.Finalization; use Ada.Finalization;
package Unbounded_String_Pkg is
   -- Implement unbounded strings
   type Ustring_T is private;
   function "=" (L, R : Ustring_T) return Boolean;
   function To_Ustring_T (Item : String) return Ustring_T;
   function To_String (Item : Ustring_T) return String;
   function Length (Item : Ustring_T) return Natural;
   function "&" (L, R : Ustring_T) return Ustring_T;
private
   type String_Ref is access String;
   type Ustring_T is new Controlled with record
      Ref : String_Ref := new String (1 .. 0);
   end record;
   procedure Finalize (Object : in out Ustring_T);
   procedure Adjust (Object : in out Ustring_T);
end Unbounded_String_Pkg;
```

Unbounded String Implementation

```
with Ada.Unchecked_Deallocation;
package body Unbounded_String_Pkg is
   procedure Free_String is new Ada.Unchecked_Deallocation
     (String, String_Ref);

   function "=" (L, R : Ustring_T) return Boolean is
      (L.Ref.all = R.Ref.all);

   function To_Ustring_T (Item : String) return Ustring_T is
      (Controlled with Ref => new String'(Item));

   function To_String (Item : Ustring_T) return String is
      (Item.Ref.all);

   function Length (Item : Ustring_T) return Natural is
      (Item.Ref.all'Length);

   function "&" (L, R : Ustring_T) return Ustring_T is
      (Controlled with Ref => new String'(L.Ref.all & R.Ref.all);

   procedure Finalize (Object : in out Ustring_T) is
   begin
      Free_String (Object.Ref);
   end Finalize;

   procedure Adjust (Object : in out Ustring_T) is
   begin
      Object.Ref := new String'(Object.Ref.all);
   end Adjust;
end Unbounded_String_Pkg;
```

Finalizable Aspect

- Uses the GNAT-specific `with Finalizable` aspect

```
type Ctrl is record
   Id : Natural := 0;
end record
  with Finalizable => (Initialize             => Initialize,
                       Adjust                 => Adjust,
                       Finalize               => Finalize,
                       Relaxed_Finalization => True);

procedure Adjust     (Obj : in out Ctrl);
procedure Finalize   (Obj : in out Ctrl);
procedure Initialize (Obj : in out Ctrl);
```

- `Initialize`, `Adjust` same definition as previously
- `Finalize` has the `No_Raise` aspect: it cannot raise exceptions
- `Relaxed_Finalization`
  - Performance on-par with C++'s destructor
  - No automatic finalization of **heap-allocated** objects

Lab

Controlled Types Lab

- Requirements
  - Create a simplistic secure key tracker system
    - Keys should be unique
    - Keys cannot be copied
    - When a key is no longer in use, it is returned back to the system
  - Interface should contain the following methods
    - Generate a new key
    - Return a generated key
    - Indicate how many keys are in service
    - Return a string describing the key
  - Create a main program to generate / destroy / print keys
- Hints
  - Need to return a key when out-of-scope OR on user request
  - Global data to track used keys

Controlled Types Lab Solution - Keys (Spec)

```
with Ada.Finalization;
package Keys_Pkg is

   type Key_T is limited private;
   function Generate return Key_T;
   procedure Destroy (Key : Key_T);
   function In_Use return Natural;
   function Image (Key : Key_T) return String;

private
   type Key_T is new Ada.Finalization.Limited_Controlled with record
      Value : Character;
   end record;
   procedure Initialize (Key : in out Key_T);
   procedure Finalize (Key : in out Key_T);

end Keys_Pkg;
```

Controlled Types Lab Solution - Keys (Body)

```
package body Keys_Pkg is
   Global_In_Use : array (Character range 'a' .. 'z') of Boolean :=
     (others => False);

   pragma Warnings (Off);
   function Next_Available return Character is
   begin
      for C in Global_In_Use'Range loop
         if not Global_In_Use (C) then
            return C;
         end if;
      end loop;
      -- we ran out of keys! exception if we get here
   end Next_Available;
   pragma Warnings (On);

   function In_Use return Natural is
      Ret_Val : Natural := 0;
   begin
      for Flag of Global_In_Use loop
         Ret_Val := Ret_Val + (if Flag then 1 else 0);
      end loop;
      return Ret_Val;
   end In_Use;

   function Generate return Key_T is
   begin
      return X : Key_T;
   end Generate;

   procedure Destroy (Key : Key_T) is
   begin
      Global_In_Use (Key.Value) := False;
   end Destroy;

   function Image (Key : Key_T) return String is
      ("KEY: " & Key.Value);

   procedure Initialize (Key : in out Key_T) is
   begin
      Key.Value                 := Next_Available;
      Global_In_Use (Key.Value) := True;
   end Initialize;

   procedure Finalize (Key : in out Key_T) is
   begin
      Global_In_Use (Key.Value) := False;
   end Finalize;
end Keys_Pkg;
```

Controlled Types Lab Solution - Main

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

   procedure Generate (Count : Natural) is
      Keys : array (1 .. Count) of Keys_Pkg.Key_T;
   begin
      Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));
      for Key of Keys
      loop
         Put_Line ("   " & Keys_Pkg.Image (Key));
      end loop;
   end Generate;

begin
   Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));

   Generate (4);
   Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));

end Main;
```

Summary

Summary

- Controlled types allow access to object construction, assignment, destruction
- **Ada.Finalization** can be expensive to use
    - Other mechanisms may be more efficient
        - But require more rigor in usage

Expert Resource Management

Indefinite Private

Limited Private

```
type T is limited private;
```

- Same interface as private
  - Removes = and /=
  - Removes assignments
  - Removes copies

Note

Private type is a **view**

- Completion should provide **at least** the same set of features
- Completion can be a `limited record`
- ... but doesn't **have** to

Limited Private

- No assignment: user cannot duplicate a key
- No equality: user cannot check two keys are the same
- Private type: user cannot access or change the issued date

```
package Key_Stuff is
   type Key is limited private;
   function Make_Key ( ... ) return Key;
   ...

package body Key_Stuff is
   function Make_Key ( ... ) return Key is
   begin
      return New_Key: Key do
         New_Key.Issued := Today;
         ...
      end return;
   end Make_Key;
```

⚠ Warning

- Definite type
- User **doesn't** have to call `Make_Key`

Indefinite Private

- Indefinite: user **must** use the constructors
- Delegated `constant` objects are static constructors

```
package Binary_Trees is
  type Tree_T (<>) is private;

  Empty_Tree : constant Tree_T;

  type Nodes_T is ...
  type Edges_T is ...
  procedure Make (Nodes : Nodes_T; Edges : Edges_T);
  ...
private
  type Tree_T is record
  ...

  Empty_Tree : constant Tree_T := ...;

end Binary_Trees;
```

Tip

Type completion **can** be definite

Opaque Pointers

- User can instantiate
- Completion is an `access`
- Concrete type being pointed to is **incomplete**
- Implementation is done entirely within the body

```
package Black_Boxes is
   type Box_T is private;
   procedure Foo (B : Box_T);
private
   type Internal_Box_T; -- incomplete
   type Box_T is access all Internal_Box_T;
end Black_Boxes;
```

Example: A String Holder (1/2)

- Implementation not discussed here

```
package String_Holders is
   type Info is limited private;

   function Contains (Obj : Info; Str : String) return Boolean
      with Ghost;
   function Equals (Left, Right : Info) return Boolean
      with Ghost;
```

> **Tip**
> These are only used for contracts, hence the Ghost aspect

```
function To_Info (Str : String) return Info
   with Post => Contains (To_Info'Result, S);

function To_String (Obj : Info)
                     return String
   with Post => Contains (Obj, To_String'Result);

procedure Copy (To   : in out Info;
                From :        Info)
   with Post => Equals (To, From);

procedure Append (Obj : in out Info;
                  Str : String)
   with Post => Contains (Obj, To_String (Obj)'Old & S);

procedure Destroy (Obj : in out Info);
```

Example: A String Holder (2/2)

```
private
   type Info is access String;

   function To_String_Internal (Obj : Info) return String
      is (if Obj = null then "" else Obj.all);
```

Tip

This can be used by contracts implementation below, and child packages

```
function Contains (Obj : Info; Str : String) return Boolean
   is (Obj /= null and then Obj.all = Str);
function Equals (Left, Right : Info) return Boolean
   is (To_String_Internal (Left)
     = To_String_Internal (Right));
```

Reference Counting Using Controlled Types

Global Overview

- Idiom for counting object references
  - Safe deallocation
  - No memory leak
  - Efficient
  - All access must then be using it
- Any refcounted type **must** derive from Refcounted
  - Tagged
  - Get a Ref through Set
  - Turn a Ref into an access through Get

```
package Ref_Counter is
   type Refcounted is abstract tagged private;
   procedure Free (Self : in out Refcounted) is null;

   type Refcounted_Access is access all Refcounted'Class;
   type Ref is tagged private;

   procedure Set (Self : in out Ref; Data : Refcounted'Class);
   function Get (Self : Ref) return Refcounted_Access;
   procedure Finalize (P : in out Ref);
   procedure Adjust   (P : in out Ref);
private
   type Refcounted is abstract tagged record
      Refcount : Integer := 0;
   end record;

   type Ref is new Ada.Finalization.Controlled with record
       Data : Refcounted_Access;
   end record;
```

Implementation Details

```
procedure Set (Self : in out Ref; Data : Refcounted'Class)
```

Tip

This procedure is safe

- Ref default value is null
- Clears up any previously used Ref

```
is
  D : constant Refcounted_Access := new Refcounted'Class'(Data);
begin
  if Self.Data /= null then
      Finalize (Self); -- decrement old reference count
  end if;

  Self.Data := D;
  Adjust (Self);  -- increment reference count (set to 1)
end Set;
```

```
overriding procedure Adjust (P : in out Ref)
```

Note

Called for all new references

Warning

Data might be null

```
is
begin
   if P.Data /= null then
      P.Data.Refcount := P.Data.Refcount + 1;
   end if;
end Adjust;
```

Logger

Public Interface

- Logger uses a file for writing
- **limited** cannot be copied, or compared
- **procedure** Put_Line for logging

```
type Logger (Filename : not null access constant String)
  is tagged limited private;

procedure Put_Line
  (L : Logger; S : String);
```

Implementation: Private part

```
type Logger (Filename : not null access constant  String)
   is new Ada.Finalization.Limited_Controlled with
```

Note

- `Limited_Controlled`
- Maintains a handle to the log file

```
record
   Logfile : Ada.Text_IO.File_Type;
end record;

procedure Initialize (L : in out Logger);
--  opens the file
procedure Finalize (L : in out Logger);
--  closes the file
```

Implementation: Body

- Trivial

Tip
Once the hard part of designing the interface is done, implementation is trivial.

```
with Ada.Text_IO; use Ada.Text_IO;

package body Loggers is
   procedure Initialize (L : in out Logger) is
   begin
        Create (L.Logfile, Out_File, L.Filename.all);
        Put_Line (L, "Starting");
   end Initialize;

   procedure Put_Line (L : Logger; S : String) is
   begin
        Put_Line ("Logger: " & S);
        Put_Line (L.Logfile, S);
   end Put_Line;

   procedure Finalize
        (L : in out Logger) is
   begin
        Put_Line (L, "Closing");
        Close (L.Logfile);
   end Finalize;
end Loggers;
```

Refcounting Wrapper for External C Objects

Context

- From https://blog.adacore.com/the-road-to-a-thick-opengl-binding-for-ada-part-2
- OpenGL API create various objects like textures or vertex buffers
- Creating them gives us an ID
    - Can then be used to refer to the object
- Simple approach: Manually reclaiming them
    - Could cause leaks
- Refcount approach: automatic ID management
    - From an Ada wrapper
    - Automatic reclaim once the last reference vanishes

Wrapper Interface

- `type GL_Object is abstract tagged private`
    - Implements smart pointer logic

```
procedure Initialize_Id (Object : in out GL_Object);

procedure Clear (Object : in out GL_Object);

function Initialized (Object : GL_Object) return Boolean;
```

- Derived by the **actual** object types

```
procedure Internal_Create_Id
  (Object : GL_Object; Id : out UInt) is abstract;

procedure Internal_Release_Id
  (Object : GL_Object; Id : UInt) is abstract;
```

- Example usage

```
type Shader (Kind : Shader_Type) is new GL_Object with null record;
```

Wrapper Implementation: Private part

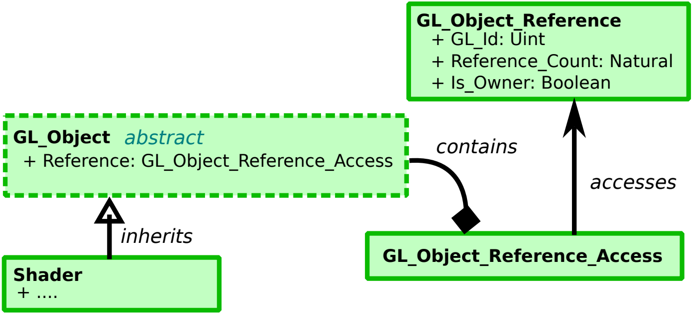
- Object ID's holder: `GL_Object_Reference`
    - All derived types have a handle to this

```
type GL_Object_Reference;
type GL_Object_Reference_Access is access all GL_Object_Reference;

type GL_Object is abstract new Ada.Finalization.Controlled
  with record
   Reference : GL_Object_Reference_Access := null;
end record;
```

- Controlled type implementing **ref-counting**

```
overriding procedure Adjust (Object : in out GL_Object);
-- Increases reference count.

overriding procedure Finalize (Object : in out GL_Object);
-- Decreases reference count.
-- Destroys underlying resource when it reaches zero.
```

Wrapper Implementation: Full Picture




```
type GL_Object_Reference is record
   GL_Id            : UInt;
   Reference_Count : Natural;
   Is_Owner         : Boolean;
end record;
```

Adjust Completion

- `Adjust` is called every time a new reference is **created**
- Increments the ref-counter

```
overriding procedure Adjust (Object : in out GL_Object) is
begin
   if Object.Reference /= null then
      Object.Reference.Reference_Count := @ + 1;
   end if;
end Adjust;
```

Finalize Completion

Note

- Finalize should always be *idempotent*
  - Compiler might call it multiple times on the same object
  - In particular when **exceptions** occur

```
overriding procedure Finalize (Object : in out GL_Object) is
   Ref : GL_Object_Reference_Access
        renames Object.Reference;
begin
```

Warning

Do **not** decrement the reference counter for every call

- A given object will own **only one** reference

```
--  Idempotence: the next call to Finalize will have no effect
Ref := null;

if Ref /= null then
   Ref.Reference_Count := @ - 1;
   if Ref.Reference_Count = 0 then
      Free (Ref.all);  --  Call to user-defined primitive
      Unchecked_Free (Ref);
   end if;
end if;
```

GNAT Semaphores

Semaphores

- Shared counters
- Multitask-safe
  - Support priorities from "Real-time Systems" LRM Annex D
- `Counting_Semaphore` and `Binary_Semaphore`
  - `protected` types
  - Counting holds an Integer
  - Binary holds a Boolean
- Priority ceiling (LRM D.3)
  - For `pragma Locking_Policy (Ceiling_Locking)`
  - Protects against priority inversions

Interface

```
protected type Counting_Semaphore
   (Initial_Value : Natural;
   [...]

   entry Seize;
   --  Blocks caller until/unless the semaphore's internal counter is
   --  greater than zero. Decrements the semaphore's internal counter u
   --  executed.

   procedure Release;
   --  Increments the semaphore's internal counter

protected type Binary_Semaphore
  (Initially_Available : Boolean;

subtype Mutual_Exclusion is Binary_Semaphore
  (Initially_Available => True,
   Ceiling             => Default_Ceiling);
```

Idiom: Scope Locks

- Automatic release

```
type Scope_Lock (Lock : access Mutual_Exclusion) is
   new Ada.Finalization.Limited_Controlled with null record;

procedure Initialize (This : in out Scope_Lock) is
begin
  This.Lock.Seize;
end Initialize;

procedure Finalize (This : in out Scope_Lock) is
begin
  This.Lock.Release;
end Finalize;

Mutex : aliased Mutual_Exclusion;

State : Integer := 0;

procedure Operation_1 is
   S : Scope_Lock (Mutex'Access);
begin
   State := State + 1;  -- for example...
   Put_Line ("State is now" & State'Img);
end Operation_1;
```

Task Safe Interfaces

Problem Statement

- Designing task-safe code requires using dedicated constructs
- How to reuse the components?
- How to refactor task-unsafe code into task-safe version?

Access Protected

- Access to `protected` objects' subprograms
- `type P is access protected procedure (args...)`
- `type F is access protected function (args...) return ...`

```
type Work_Id is tagged limited private;

type Work_Handler is
   access protected procedure (T : Work_Id);
```

Synchronized Interface

- **synchronized interface** can be inherited by **task**/**protected** types

```
type Counter_I is synchronized interface;
procedure Increment (Counter : in out Counter_I) is abstract;

task type Counter_Task_T is new Counter_I with
   -- Always implemented as an entry for tasks
   entry Increment;
end task;

protected type Counter_Prot_T is new Counter_I with
   procedure Increment;
end Counter_Prot_T;
```

- Also present:
  - **task interface** meant for tasks only
  - **protected interface** meant for protected types only

> ⚠ **Warning**
>
> Only available in **full-tasking** runtimes

Standard Library Queues Interface

- In Ada.Containers
- Synchronized_Queue_Interfaces interface

Tip

Provides a portable interface

```
generic
   type Element_Type is private;
package Ada.Containers.Synchronized_Queue_Interfaces is
   type Queue is synchronized interface;
```

Standard Library Queues Implementations

- Four implementations

Tip
Recommended over rolling-out one's own queue implementation

- Synchronized implementations
  - `Unbounded_Synchronized_Queues`
  - `Bounded_Synchronized_Queues`
  - As `protected` types
  - With priority ceiling
- Priority implementations
  - `Unbounded_Priority_Queues`
  - `Bounded_Priotiry_Queues`
  - As `protected` types
  - Elements provide `Get_Priority`
    - Used for sorting elements

Example: Scheduler Interface

```
type Scheduler_I;
type Maybe_Work_Item_I is access protected procedure;
type Work_Item_I is not null access protected procedure;

type Scheduler_I is synchronized interface;
procedure Queue (S : in out Scheduler_I; W : Work_Item_I) is abstract;
procedure Execute_Next (S : in out Scheduler_I) is abstract;

type Work_Items_Array is array (Positive range <>)
  of Maybe_Work_Item_I;

protected type Scheduler_T (Size : Positive) is new Scheduler_I with
   procedure Queue (W : Work_Item_I);
   entry Execute_Next;
private
   Number_Of_Items : Natural := 0;
   Items : Work_Items_Array (1 .. Size);
end Scheduler_T;
```

Example: Scheduler (Body)

```
protected body Scheduler_T is
   procedure Queue (W : Work_Item_I) is
   begin
      Number_Of_Items := Number_Of_Items + 1;
      Items (Number_Of_Items) := Maybe_Work_Item_I (W);
   end Queue;

   entry Execute_Next
      when Number_Of_Items > 0
   is
      W : Work_Item_I := Work_Item_I (Items (Number_Of_Items));
   begin
      Number_Of_Items := Number_Of_Items - 1;
      W.all;
   end Execute_Next;
end Scheduler_T;
```

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

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

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

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

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

Declaration

- Subprograms

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

- Packages

```
generic
   type T is private;
package Stack is
   procedure Push (Item : T);
end Stack;
```

- Body is required
  - Will be specialized and compiled for **each instance**
- Children of generic units have to be generic themselves

```
generic
package Stack.Utilities is
   procedure Print (S : Stack_T);
```

Usage

- Instantiated with the **new** keyword

```
--  Standard library
function Convert is new Ada.Unchecked_Conversion
  (Integer, Array_Of_4_Bytes);
--  Callbacks
procedure Parse_Tree is new Tree_Parser
  (Visitor_Procedure);
--  Containers, generic data-structures
package Integer_Stack is new Stack (Integer);
```

- Advanced usages for testing, proof, meta-programming

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<code goes here>
```

A. `package`
B. `record`
C. `function`
D. `array`

Quiz

Which one(s) of the following can be made generic?

```
generic
   type T is private;
<code goes here>
```

A. *package*
B. record
C. *function*
D. array

Only packages, functions, and procedures, can be made generic.

Generic Data

Generic Types Parameters (1/3)

- 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;`	Incomplete type; can only be used as target of `access`
`type T5 is tagged private;`	`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 Component (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_Component is new Component
   (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 (are) legal instantiation(s)?

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 (are) legal instantiation(s)?

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 Component_T is private;
  Array_Size     : Positive;
  High_Watermark : in out Component_T;
package Repository is
```

- Generic instance

```
V   : Float;
Max : Float;

procedure My_Repository is new Repository
  (Component_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

- **is** <> - matching subprogram is taken by default
- **is null** - null procedure 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 procedure
```

Quiz

```
generic
   type Component_T is (<>);
   Last : in out Component_T;
procedure Write (P : Component_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 (Numeric))`
D. `procedure Write_D is new Write (Float, Floating_Point)`

Quiz

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

Given the following generic function:

```
generic
   type Some_T is private;
   with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
   return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
   Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
   ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

A. `function IncrA is new Incr (Integer, Weird);`
B. `function IncrB is new Incr (Record_T, Add);`
C. `function IncrC is new Incr (Record_T);`
D. `function IncrD is new Incr (Integer);`

Quiz

Given the following generic function:

```
generic
   type Some_T is private;
   with function "+" (L : Some_T; R : Integer) return Some_T is <>;
function Incr (Param : Some_T) return Some_T;

function Incr (Param : Some_T) return Some_T is
begin
   return Param + 1;
end Incr;
```

And the following declarations:

```
type Record_T is record
   Component : Integer;
end record;
function Add (L : Record_T; I : Integer) return Record_T is
   ((Component => L.Component + I))
function Weird (L : Integer; R : Integer) return Integer is (0);
```

Which of the following instantiation(s) is/are **not** legal?

A. `function IncrA is new Incr (Integer, Weird);`
B. `function IncrB is new Incr (Record_T, Add);`
C. *`function IncrC is new Incr (Record_T);`*
D. `function IncrD is new Incr (Integer);`

`with function "+" (L : Some_T; R : Integer) return Some_T is <>;` indicates that if no function for + is passed in, find (if possible) a matching definition at the point of instantiation.

A. `Weird` matches the subprogram profile, so `Incr` will use `Weird` when doing addition for `Integer`
B. `Add` matches the subprogram profile, so `Incr` will use `Add` when doing the addition for `Record_T`
C. There is no matching + operation for `Record_T`, so that instantiation fails to compile
D. Because there is no parameter for the generic formal parameter +, the compiler will look for one in the scope of the instantiation. Because the instantiating type is numeric, the inherited + operator is found

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)`

Genericity Lab

- Requirements
  - Create a record structure containing multiple components
    - 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 components
  - Print routine will need to know how to print one component

Genericity Lab Solution - Generic (Spec)

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

   type List_T is private;

   procedure Add (This : in out List_T;
                  Item : in     Component_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 Component_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     Component_T) is
   begin
      This.Length               := This.Length + 1;
      This.Values (This.Length) := Item;
   end Add;

   procedure Sort (This : in out List_T) is
      Temp : Component_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 (Component_T => Data_Type.Record_T,
                                     Max_Size  => 20,
                                     ">"       => Data_Type.">",
                                     Image => Data_Type.Image);

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

begin
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'a'));
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'z'));
   List.Add (My_List, (Integer_Component   => 111,
                       Character_Component => 'A'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'B'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'Y'));
   List.Add (My_List, (Integer_Component   => 999,
                       Character_Component => 'b'));
   List.Add (My_List, (Integer_Component   => 112,
                       Character_Component => 'a'));
   List.Add (My_List, (Integer_Component   => 998,
                       Character_Component => '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 run-time** depending on the type at call-site
- Types can be **extended** by other packages
  - Conversion 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

- 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_T is new Shape_T with record
   Origin : Coord_T;
end Point_T;

-- 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

- 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(s) is (are) legal?

A.
```
type A1 is record
  Component1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Component2 : Integer;
end record;
type B2 is new B1 with
record
  Component2b :
Integer;
end record;
```

C.
```
type C1 is tagged
record
  Component3 : Integer;
end record;
type C2 is new C1 with
record
  Component3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Component1 : Integer;
end record;
type D2 is new D1;
```

Quiz

Which code block(s) is (are) legal?

A.
```
type A1 is record
  Component1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Component2 : Integer;
end record;
type B2 is new B1 with
record
  Component2b :
Integer;
end record;
```

C.
```
type C1 is tagged
record
  Component3 : Integer;
end record;
type C2 is new C1 with
record
  Component3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Component1 : 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 components
  - We can extend a **tagged** type by adding components

```
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 components (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 components 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 components
  - 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" components 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 Create 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 Create 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` component, 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

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 components, 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 components
- 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

Polymorphism

Introduction

Introduction

- `'Class` operator to categorize *classes of types*
- Type classes allow dispatching calls
  - Abstract types
  - Abstract subprograms
- Runtime call dispatch vs compile-time call dispatching

Classes of Types

Classes

- In Ada, a Class denotes an inheritance subtree
- Class of **Root** is the class of **Root** and all its children
- Type `Root'Class` can designate any object typed after type of class of **Root**

```
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 `Root'Class` have at least the properties of **Root**
  - Components of **Root**
  - Primitives of **Root**

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 Animal is tagged null record;
   type Dog is new Animal with null record;
   procedure Handle_Animal (Some_Animal : in out Animal'Class) is null;
   My_Dog     : Dog;
   Pet        : Dog'Class    := My_Dog;
   Pet_Animal : Animal'Class := Pet;
   Pet_Dog    : Animal'Class := My_Dog;
   -- initialization required in class-wide declaration
   Bad_Animal : Animal'Class; -- compile error
   Bad_Dog    : Dog'Class;    -- compile error
begin
   Handle_Animal (Pet);
   Handle_Animal (My_Dog);
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

```ada
type Animal is abstract tagged record
   Number_Of_Eyes : Integer;
end record;
procedure Feed (The_Animal : Animal) is abstract;
procedure Pet (The_Animal : Animal);
type Dog is abstract new Animal with null record;
type Bulldog is new Dog with null record;

overriding  -- Ada 2005 and later
procedure Feed (The_Animal : Bulldog);
```

- C++

```cpp
class Animal {
   public:
      int Number_Of_Eyes;
      virtual void Feed (void) = 0;
      virtual void Pet (void);
};
class Dog : public Animal {
};
class Bulldog {
   public:
      virtual void Feed (void);
};
```

Relation to Primitives

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

```
type Root is tagged null record;
procedure Not_A_Primitive (Param : Root'Class);
type Child is new Root with null record;
-- This does not override Not_A_Primitive!
overriding procedure Not_A_Primitive (Param : Child'Class);
```

'Class and Prefix Notation

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

```
type Animal is tagged null record;
procedure Handle_Animal (Some_Animal : Animal'Class);
type Cat is new Animal with null record;

Stray_Animal : Animal;
Pet_Animal   : Animal'Class := Animal'(others => <>);
...
Handle_Animal (Stray_Animal);
Handle_Animal (Pet_Animal);
Stray_Animal.Handle_Animal;
Pet_Animal.Handle_Animal;
```

Dispatching and Redispatching

Calls on Class-Wide Types (1/3)

- Any subprogram expecting a **Root** object can be called with a `Animal'Class` object

```
type Animal is tagged null record;
procedure Feed (The_Animal : Animal);

type Dog is new Animal with null record;
procedure Feed (The_Dog : Dog);

   Stray_Dog : Animal'Class := [...]
   My_Dog    : Dog'Class := [...]
begin
   Feed (Stray_Dog);
   Feed (My_Dog);
```

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 run-time

Ada

```
declare
  Stray : Animal'Class :=
      Animal'(others => <>);
  My_Dog : Animal'Class :=
      Dog'(others => <>);
begin
  Stray.Feed;  -- calls Feed of Animal
  My_Dog.Feed; -- calls Feed of Dog
```

C++

```
Animal * Stray =
            new Animal ();
Animal * My_Dog = new Dog ();
Stray->Feed ();
My_Dog->Feed ();
```

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
  Stray : Animal'Class :=
       Animal'(others => <>);
  My_Dog : Animal'Class :=
       Dog'(others => <>);
begin
  Animal (Stray).Feed;  -- calls Feed of Animal
  Animal (My_Dog).Feed; -- calls Feed of Animal
```

C++

```cpp
Animal * Stray =
            new Animal ();
Animal * My_Dog = new Dog ();
((Animal) *Stray).Feed ();
((Animal) *My_Dog).Feed ();
```

Definite and Class-Wide Views

- In C++, dispatching occurs only on pointers
- In Ada, dispatching occurs only on class-wide views

```
type Animal is tagged null record;
procedure Groom (The_Animal : Animal);
procedure Give_Treat (The_Animal : Animal);
type Dog is new Animal with null record;
overriding procedure Give_Treat (The_Dog : Dog);
procedure Groom (The_Animal : Animal) is
begin
   Give_Treat (The_Animal); -- always calls Give_Treat from Animal
end Groom;
procedure Main is
   My_Dog : Animal'Class :=
        Dog'(others => <>);
begin
   -- Calls Groom from the implicitly overridden subprogram
   -- Calls Give_Treat from Animal!
   My_Dog.Groom;
```

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 Animal is tagged null record;
procedure Feed (An_Animal : Animal);
procedure Pet (An_Animal : Animal);
type Cat is new Animal with null record;
overriding procedure Pet (A_Cat : Cat);
```

Redispatching Example

```
procedure Feed (Anml : Animal) is
   Fish : Animal'Class renames
            Animal'Class (Anml); -- naming of a view
begin
   Pet (Anml); -- static: uses the definite view
   Pet (Animal'Class (Anml)); -- dynamic: (redispatching)
   Pet (Fish);                -- dynamic: (redispatching)

   -- Ada 2005 "distinguished receiver" syntax
   Anml.Pet; -- static: uses the definite view
   Animal'Class (Anml).Pet; -- dynamic: (redispatching)
   Fish.Pet;                -- dynamic: (redispatching)
end Feed;
```

Quiz

```
package Robots is
   type Robot is tagged null record;
   function Service_Code (The_Bot : Robot) return Integer is (101);
   type Appliance_Robot is new Robot with null record;
   function Service_Code (The_Bot : Appliance_Robot) return Integer is (201);
   type Vacuum_Robot is new Appliance_Robot with null record;
   function Service_Code (The_Bot : Vacuum_Robot) return Integer is (301);
end Robots;

with Robots; use Robots;
procedure Main is
   Robot_Object : Robot'Class := Vacuum_Robot'(others => <>);
```

What is the value returned by
`Service_Code (Appliance_Robot'Class (Robot_Object));`?

A. 301
B. 201
C. 101
D. Compilation error

Quiz

```
package Robots is
   type Robot is tagged null record;
   function Service_Code (The_Bot : Robot) return Integer is (101);
   type Appliance_Robot is new Robot with null record;
   function Service_Code (The_Bot : Appliance_Robot) return Integer is (201);
   type Vacuum_Robot is new Appliance_Robot with null record;
   function Service_Code (The_Bot : Vacuum_Robot) return Integer is (301);
end Robots;

with Robots; use Robots;
procedure Main is
   Robot_Object : Robot'Class := Vacuum_Robot'(others => <>);
```

What is the value returned by
`Service_Code (Appliance_Robot'Class (Robot_Object));`?

A. ***301***
B. 201
C. 101
D. Compilation error

Explanations

A. Correct
B. Would be correct if `Robot_Object` was a `Appliance_Robot` - `Appliance_Robot'Class` leaves the object as `Vacuum_Robot`
C. Object is initialized to something in `Robot'Class`, but it doesn't have to be `Robot`
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 Animal is tagged null record;
procedure Interact (Left : Animal; Right : Animal);
type Dog is new Animal with null record;
overriding procedure Interact (Left : Dog; Right : Dog);
```

- At call time, all actual parameters' tags have to match, either statically or dynamically

```
Animal_1, Animal_2   : Animal;
Dog_1, Dog_2 : Dog;
Any_Animal_1 : Animal'Class := Animal_1;
Any_Animal_2 : Animal'Class := Animal_2;
Dog_Animal : Animal'Class := Dog_1;
...
Interact (Animal_1, Animal_2);                         -- static:  ok
Interact (Animal_1, Dog_1);                            -- static:  error
Interact (Any_Animal_1, Any_Animal_2);                 -- dynamic: ok
Interact (Any_Animal_1, Dog_Animal);                   -- dynamic: error
Interact (Animal_1, Any_Animal_1);                     -- static:  error
Interact (Animal'Class (Animal_1), Any_Animal_1); -- 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 Animal is tagged null record;
function "=" (Left : Animal; Right : Animal) return Boolean;
type Dog is new Animal with null record;
overriding function "=" (Left : Dog; Right : Dog) return Boolean;
Animal_1, Animal_2 : Animal;
Dog_1, Dog_2 : Child;
Any_Animal_1 : Animal'Class := Animal_1;
Any_Animal_2 : Animal'Class := Animal_2;
Dog_Animal   : Animal'Class := Dog_1;
...
-- overridden "=" called via dispatching
if Any_Animal_1 = Any_Animal_2 then [...]
if Any_Animal_1 = Dog_Animal then [...] -- returns false
```

Controlling Result (1/2)

- The controlling operand may be the return type
  - This is known as the constructor pattern

```
type Animal is tagged null record;
function Feed_Treats (Number_Of_Treats : Integer) return Animal;
```

- If the child adds components, all such subprograms have to be overridden

```
type Animal is tagged null record;
function Feed_Treats (Number_Of_Treats : Integer) return Animal;

type Dog is new Animal with null record;
--  OK, Feed_Treats is implicitly inherited

type Bulldog is new Animal with record
   Has_Underbite : Boolean;
end record;
--  ERROR no implicitly inherited function Feed_Treats
```

- Primitives returning abstract types have to be abstract

```
type Animal is abstract tagged null record;
function Feed_Treats (Number_Of_Treats : Integer) return Animal is abstract;
```

Controlling Result (2/2)

- Primitives returning **tagged** types can be used in a static context

```
type Animal is tagged null record;
function Feed return Animal;
type Dog is new Animal with null record;
function Feed return Dog;
Fed_Animal : Animal := Feed;
```

- In a dynamic context, the type has to be known to correctly dispatch

```
Fed_Animal : Animal'Class :=
                      Animal'(Feed);    -- Static call to Animal primitive
Another_Fed_Animal : Animal'Class := Fed_Animal;
Fed_Dog : Animal'Class := Dog'(Feed);   -- Static call to Dog primitive
Starving_Animal : Animal'Class := Feed; -- Error - ambiguous expression
...
Fed_Animal := Feed;            -- Dispatching call to Animal primitive
Another_Fed_Animal := Feed; -- Dispatching call to Animal primitive
Fed_Dog := Feed;               -- Dispatching call to Dog primitive
```

- No dispatching is possible when returning access types

Lab

Polymorphism Lab

- Requirements
  - Create a multi-level types hierarchy of shapes
    - Level 1: Shape → Quadrilateral | Triangle
    - Level 2: Quadrilateral → 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` attribute
  - 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
- Runtime call dispatch vs compile-time call dispatching
  - Compiler resolves appropriate call where it can
  - Runtime resolves appropriate call where it can
  - If not resolved, exception

Exceptions In-Depth

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
- 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];
```

  Note "with string_expression" only available in Ada 2005 and later
- A `raise` by itself is only allowed in handlers

```ada
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;
```

Language-Defined Exceptions

Constraint_Error

- Caused by violations of constraints on range, index, etc.
- The most common exceptions encountered

```
K : Integer range 1 .. 10;
...
K := -1;
```

```
L : array (1 .. 100) of Some_Type;
...
L (400) := SomeValue;
```

Program_Error

- When runtime control structure is violated
    - Elaboration order errors and function bodies
- When implementation detects bounded errors
    - Discussed momentarily

```
function F return Some_Type is
begin
  if something then
    return Some_Value;
  end if; -- program error - no return statement
end F;
```

Storage_Error

- When insufficient storage is available
- Potential causes
    - Declarations
    - Explicit allocations
    - Implicit allocations

```
Data : array (1..1e20) of Big_Type;
```

Explicitly-Raised Exceptions

- Raised by application via `raise` statements
  - Named exception becomes active
- Syntax

```
raise_statement ::= raise; |
   raise exception_name
   [with string_expression];
```

  - `with string_expression` only available in Ada 2005 and later
- A `raise` by itself is only allowed in handlers (more later)

```
if Unknown (User_ID) then
  raise Invalid_User;
end if;

if Unknown (User_ID) then
  raise Invalid_User
     with "Attempt by " &
          Image (User_ID);
end if;
```

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

```ada
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)

Partial and Nested Handlers

Partially Handling Exceptions

- Handler eventually re-raises the current exception
- Achieved using **raise** by itself, since re-raising
  - Current active exception is then propagated to caller

```
procedure Joy_Ride is
   ...
begin
  while not Bored loop
    Steer_Aimlessly (Bored);
    Consume_Fuel (Hot_Rod);
  end loop;
exception
  when Fuel_Exhausted =>
    Pull_Over;
    raise; -- no gas available
end Joy_Ride;
```

Typical Partial Handling Example

- Log (or display) the error and re-raise to caller
    - Same exception or another one

```
procedure Get (Item : out Integer;   From : in File) is
begin
  Ada.Integer_Text_IO.Get (From, Item);
exception
  when Ada.Text_IO.End_Error =>
    Display_Error ("Attempted read past end of file");
    raise Error;
  when Ada.Text_IO.Mode_Error =>
    Display_Error ("Read from file opened for writing");
    raise Error;
  when Ada.Text_IO.Status_Error =>
    Display_Error ("File must be opened prior to use");
    raise Error;
  when others =>
    Display_Error ("Error in Get (Integer) from file");
    raise;
end Get;
```

Exceptions Raised During Elaboration

- I.e., those occurring before the **begin**
- Go immediately to the caller
- No handlers in that frame are applicable
  - Could reference declarations that failed to elaborate!

```
procedure P (Output : out BigType) is
  -- storage error handled by caller
  N : array (Positive) of BigType;
  ...
begin
  ...
exception
  when Storage_Error =>
    -- failure to define N not handled here
    Output := N (1); -- if it was, this wouldn't work
    ...
end P;
```

Handling Elaboration Exceptions

```
procedure Test is
  procedure P is
    X : Positive := 0;  -- Constraint_Error!
  begin
    ...
  exception
    when Constraint_Error =>
      Ada.Text_IO.Put_Line ("Got it in P");
  end P;
begin
  P;
exception
  when Constraint_Error =>
    Ada.Text_IO.Put_Line ("Got Constraint_Error in Test");
end Test;
```

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Exception_Test (Input_Value : Integer) is
   Known_Problem : exception;
   function F (P : Integer) return Integer is
   begin
      if P > 0 then
         return P * P;
      end if;
   exception
      when others => raise Known_Problem;
   end F;
   procedure P (X : Integer) is
      A : array (1 .. F (X)) of Float;
   begin
      A := (others => 0.0);
   exception
      when others => raise Known_Problem;
   end P;
begin
   P (Input_Value);
   Put_Line ("Success");
exception
   when Known_Problem => Put_Line ("Known problem");
   when others => Put_Line ("Unknown problem");
end Exception_Test;
```

What will get printed for these values of Input_Value?

A. Integer'Last
B. Integer'First
C. 10000
D. 100

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Exception_Test (Input_Value : Integer) is
   Known_Problem : exception;
   function F (P : Integer) return Integer is
   begin
      if P > 0 then
         return P * P;
      end if;
   exception
      when others => raise Known_Problem;
   end F;
   procedure P (X : Integer) is
      A : array (1 .. F (X)) of Float;
   begin
      A := (others => 0.0);
   exception
      when others => raise Known_Problem;
   end P;
begin
   P (Input_Value);
   Put_Line ("Success");
exception
   when Known_Problem => Put_Line ("Known problem");
   when others => Put_Line ("Unknown problem");
end Exception_Test;
```

What will get printed for these values of Input_Value?

A.	Integer'Last	Known Problem
B.	Integer'First	Unknown Problem
C.	10000	Unknown Problem
D.	100	Success

Explanations

A → When `F` is called with a large `P`, its own exception handler captures the exception and raises `Constraint_Error` (which the main exception handler processes)

B/C → When the creation of `A` fails (due to `Program_Error` from passing `F` a negative number or `Storage_Error` from passing `F` a large number), then `P` raises an exception during elaboration, which is propagated to `Main`

Exceptions Raised in Exception Handlers

- Go immediately to caller unless also handled
- Goes to caller in any case, as usual

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

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

Example Propagation Beyond Scope

```
package P is
  procedure Q;
end P;
package body P is
  Error : exception;
  procedure Q is
  begin
    ...
    raise Error;
  end Q;
end P;
```

```
with P;
procedure Client is
begin
  P.Q;
exception
  -- not visible
  when P.Error =>
    ...
  -- captured here
  when others =>
    ...
end Client;
```

Mechanism to Treat Exceptions 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)

User Subprogram Parameter Example

```
with Ada.Exceptions; use Ada.Exceptions;
procedure Display_Exception
    (Error : in Exception_Occurrence)
is
  Msg : constant String := Exception_Message (Error);
  Info : constant String := Exception_Information (Error);
begin
  New_Line;
  if Info /= "" then
    Put ("Exception information => ");
    Put_Line (Info);
  elsif Msg /= "" then
    Put ("Exception message => ");
    Put_Line (Msg);
  else
    Put ("Exception name => ");
    Put_Line (Exception_Name (Error));
  end if;
end Display_Exception;
```

Exception Identity

- Attribute 'Identity converts exceptions to the type

```
package Ada.Exceptions is
   ...
   type Exception_Id is private;
   ...
   procedure Raise_Exception (E : in Exception_Id;
                              Message : in String := "");
   ...
end Ada.Exceptions;
```

- Primary use is raising exceptions procedurally

```
Foo : exception;
...
Ada.Exceptions.Raise_Exception (Foo'Identity,
                                Message => "FUBAR!");
```

Re-Raising Exceptions Procedurally

- Typical **raise** mechanism

```
begin
   ...
exception
  when others =>
    Cleanup;
    raise;
end;
```

- Procedural **raise** mechanism

```
begin
   ...
exception
  when X : others =>
    Cleanup;
    Ada.Exceptions.Reraise_Occurrence (X);
end;
```

Copying **Exception_Occurrence** Objects

- Via procedure **Save_Occurrence**
  - No assignment operation since is a `limited` type

```
Error : Exception_Occurrence;

begin
   ...
exception
   when X : others =>
      Cleanup;
      Ada.Exceptions.Save_Occurrence (X, Target => Error);
end;
```

Re-Raising Outside Dynamic Call Chain

```
procedure Demo is
  package Exceptions is new
      Limited_Ended_Lists (Exception_Occurrence,
                           Save_Occurrence);
  Errors : Exceptions.List;
  Iteration : Exceptions.Iterator;
  procedure Normal_Processing
      (Troubles : in out Exceptions.List) is ...
begin
  Normal_Processing (Errors);
  Iteration.Initialize (Errors);
  while Iteration.More loop
    declare
      Next_Error : Exception_Occurrence;
    begin
      Iteration.Read (Next_Error);
      Put_Line (Exception_Information (Next_Error));
      if Exception_Identity (Next_Error) =
         Trouble.Fatal_Error'Identity
      then
        Reraise_Occurrence (Next_Error);
      end if;
    end;
  end loop;
  Put_Line ("Done");
end Demo;
```

Raise Expressions

Raise Expressions

- **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

Fulfill Interface Promises to Clients

- If handled and not re-raised, normal processing continues at point of client's call
- Hence caller expectations must be satisfied

```
procedure Get (Reading : out Sensor_Reading) is
begin
  ...
  Reading := New_Value;
  ...
exceptions
  when Some_Error =>
    Reading := Default_Value;
end Get;

function Foo return Some_Type is
begin
  ...
  return Determined_Value;
  ...
exception
  when Some_Error =>
    return Default_Value; -- error if this isn't here
end Foo;
```

Allow Clients to Avoid Exceptions

- Callee

```
package Stack is
  Overflow : exception;
  Underflow : exception;
  function Full return Boolean;
  function Empty return Boolean;
  procedure Push (Item : in Some_Type);
  procedure Pop (Item : out Some_Type);
end Stack;
```

- Caller

```
if not Stack.Empty then
  Stack.Pop (...);  -- will not raise Underflow
```

You Can Suppress Run-Time Checks

- Syntax (could use a compiler switch instead)

```
pragma Suppress (check-name [, [On =>] name]);
```

- Language-defined checks emitted by compiler
- Compiler may ignore request if unable to comply
- Behavior will be unpredictable if exceptions occur
  - Raised within the region of suppression
  - Propagated into region of suppression

```
pragma Suppress (Range_Check);
pragma Suppress (Index_Check, On => Table);
```

Error Classifications

- Some errors must be detected at run-time
  - Corresponding to the predefined exceptions
- **Bounded Errors**
  - Need not be detected prior to/during execution if too hard
  - If not detected, range of possible effects is bounded
    - Possible effects are specified per error
  - Example: evaluating an un-initialized scalar variable
  - It might "work"!
- **Erroneous Execution**
  - Need not be detected prior to/during execution if too hard
  - If not detected, range of possible effects is not bounded
  - Example: Occurrence of a suppressed check

Lab

Exceptions In-Depth Lab

(Simplified) Calculator

- Overview
  - Create an application that allows users to enter a simple calculation and get a result
- Goal
  - Application should allow user to add, subtract, multiply, and divide
  - We want to track exceptions without actually "interrupting" the application
  - When the user has finished entering data, the application should report the errors found

Project Requirements

- Exception Tracking
  - Input errors should be flagged (e.g. invalid operator, invalid numbers)
  - Divide by zero should be it's own special case exception
  - Operational errors (overflow, etc) should be flagged in the list of errors
- Driver
  - User should be able to enter a string like "1 + 2" and the program will print "3"
  - User should not be interrupted by error messages
  - When user is done entering data, print all errors (raised exceptions)
- Extra Credit
  - Allow multiple operations on a line

Exceptions In-Depth Lab Solution - Calculator (Spec)

```
package Calculator is
   Formatting_Error : exception;
   Divide_By_Zero   : exception;
   type Integer_T is range -1_000 .. 1_000;
   function Add
     (Left, Right : String)
      return Integer_T;
   function Subtract
     (Left, Right : String)
      return Integer_T;
   function Multiply
     (Left, Right : String)
      return Integer_T;
   function Divide
     (Top, Bottom : String)
      return Integer_T;
end Calculator;
```

Exceptions In-Depth Lab Solution - Main

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;
with Calculator;            use Calculator;
with Debug_Pkg;
with Input;                 use Input;
procedure Main is
   Illegal_Operator : exception;
   procedure Parser
     (Str      :     String;
      Left     : out Unbounded_String;
      Operator : out Unbounded_String;
      Right    : out Unbounded_String) is
      I : Integer := Str'First;
   begin
      while I <= Str'Length and then Str (I) /= ' ' loop
         Left := Left & Str (I);
         I    := I + 1;
      end loop;
      while I <= Str'Length and then Str (I) = ' ' loop
         I := I + 1;
      end loop;
      while I <= Str'Length and then Str (I) /= ' ' loop
         Operator := Operator & Str (I);
         I        := I + 1;
      end loop;
      while I <= Str'Length and then Str (I) = ' ' loop
         I := I + 1;
      end loop;
      while I <= Str'Length and then Str (I) /= ' ' loop
         Right := Right & Str (I);
         I     := I + 1;
      end loop;
   end Parser;
begin
   loop
      declare
         Left, Operator, Right : Unbounded_String;
         Input                 : constant String := Get_String ("Sequence");
      begin
         exit when Input'Length = 0;
         Parser (Input, Left, Operator, Right);
         case Component (Operator, 1) is
            when '+' =>
               Put_Line
                 ("  => " &
                  Integer_T'Image (Add (To_String (Left), To_String (Right))));
            when '-' =>
               Put_Line
                 ("  => " &
                  Integer_T'Image
                    (Subtract (To_String (Left), To_String (Right))));
            when '*' =>
               Put_Line
                 ("  => " &
                  Integer_T'Image
                    (Multiply (To_String (Left), To_String (Right))));
            when '/' =>
               Put_Line
                 ("  => " &
                  Integer_T'Image
                    (Divide (To_String (Left), To_String (Right))));
            when others =>
               raise Illegal_Operator;
         end case;
      exception
         when The_Err : others =>
            Debug_Pkg.Save_Occurrence (The_Err);
      end;
   end loop;
   Debug_Pkg.Print_Exceptions;
end Main;
```

Exceptions In-Depth Lab Solution - Calculator (Body)

```
package body Calculator is
   function Value
     (Str : String)
      return Integer_T is
   begin
      return Integer_T'Value (Str);
   exception
      when Constraint_Error =>
         raise Formatting_Error;
   end Value;
   function Add
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) + Value (Right);
   end Add;
   function Subtract
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) - Value (Right);
   end Subtract;
   function Multiply
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) * Value (Right);
   end Multiply;
   function Divide
     (Top, Bottom : String)
      return Integer_T is
   begin
      if Value (Bottom) = 0 then
         raise Divide_By_Zero;
      else
         return Value (Top) / Value (Bottom);
      end if;
   end Divide;
end Calculator;
```

Exceptions In-Depth Lab Solution - Debug

```
with Ada.Exceptions;
package Debug_Pkg is
   procedure Save_Occurrence (X : Ada.Exceptions.Exception_Occurrence);
   procedure Print_Exceptions;
end Debug_Pkg;

with Ada.Exceptions;
with Ada.Text_IO;
use type Ada.Exceptions.Exception_Id;
package body Debug_Pkg is
   Exceptions     : array (1 .. 100) of Ada.Exceptions.Exception_Occurrence;
   Next_Available : Integer := 1;
   procedure Save_Occurrence (X : Ada.Exceptions.Exception_Occurrence) is
   begin
      Ada.Exceptions.Save_Occurrence (Exceptions (Next_Available), X);
      Next_Available := Next_Available + 1;
   end Save_Occurrence;
   procedure Print_Exceptions is
   begin
      for I in 1 .. Next_Available - 1 loop
         declare
            E    : Ada.Exceptions.Exception_Occurrence renames Exceptions (I);
            Flag : Character := ' ';
         begin
            if Ada.Exceptions.Exception_Identity (E) =
              Constraint_Error'Identity
            then
               Flag := '*';
            end if;
            Ada.Text_IO.Put_Line
              (Flag & " " & Ada.Exceptions.Exception_Information (E));
         end;
      end loop;
   end Print_Exceptions;
end Debug_Pkg;
```

Summary

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;
```

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
- Re-raising exceptions is a typical scenario
- Suppressing checks is allowed but requires care
  - Testing only proves presence of errors, not absence
  - Exceptions may occur anyway, with unpredictable effects

Tasking

Introduction

Concurrency Mechanisms

- Task
  - **Active**
  - Rendezvous: **Client / Server** model
  - Server **entries**
  - Client **entry calls**
  - Typically maps to OS threads
- Protected object
  - **Passive**
  - *Monitors* protected data
  - **Restricted** set of operations
  - Concurrency-safe **semantics**
  - No thread overhead
  - Very portable
- Object-Oriented
  - `Synchronized` interfaces
  - Protected objects inheritance

A Simple Task

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

```
procedure Main is
   task type Simple_Task_T;
   task body Simple_Task_T is
   begin
      loop
         delay 1.0;
         Put_Line ("T");
      end loop;
   end Simple_Task_T;
   Simple_Task : Simple_Task_T;
   -- This task starts when Simple_Task is elaborated
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

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;

T : 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
```

Rendezvous with a Task

- `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. Run-time 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. Run-time 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. Run-time 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. Run-time error
C. The calling task hangs
D. ***My_Task hangs***

Quiz

```ada
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;
   Task_Instance : T;
begin
   Task_Instance.Hello;
   Task_Instance.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;
   Task_Instance : T;
begin
   Task_Instance.Hello;
   Task_Instance.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 run-time 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. Run-time 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. Run-time 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;
```

Discriminated Protected or Task types

- Discriminant can be an `access` or discrete type
- Resulting type is indefinite
  - Unless mutable
- Example: counter shared between tasks

```
protected type Counter_T is
   procedure Increment;
end Counter_T

task type My_Task (Counter : not null access Counter_T) is [...]

task body My_Task is
begin
   Counter.Increment;
   [...]
```

Using discriminant for Real-Time aspects

```
protected type Protected_With_Priority (Prio : System.Priori
   with Priority => Prio
is
```

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. Run-time 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. Run-time error

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

Some Advanced Concepts

Waiting with a Delay

- A **select** statement may **time-out** using **delay** or **delay until**
  - Resume execution at next statement
- Multiple **delay** allowed
  - Useful when the value is not hard-coded

```
loop
  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;
end loop;
```

Task will wait up to 50 seconds for `Receive_Message`*. If no message is received, it will write to the console, and then restart the loop. (If the* `exit` *wasn't there, the loop would exit the first time no message was received.)*

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;
```

Procedure will wait up to 50 seconds for `Receive_Message` *to be accepted before it gives up*

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;

   Task_Instance : T;
begin
   delay 10.0;
   abort Task_Instance;
end;
```

`select` ... `then abort`

- `select` can call `abort`
- Can abort anywhere in the processing
- **Highly** unsafe

Multiple Select Example

```
loop
   select
      accept Receive_Message (V : String) do
         Put_Line ("Select_Loop_Task Receive: " & V);
      end Receive_Message;
   or
      accept Send_Message (V : String) do
         Put_Line ("Select_Loop_Task Send: " & V);
      end Send_Message;
   or when Termination_Flag =>
      accept Stop;
   or
      delay 0.5;
      Put_Line
        ("No more waiting at" & Day_Duration'Image (Seconds (Clock)));
      exit;
   end select;
end loop;
```

Example: Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Task_Select; use Task_Select;

procedure Main is
begin
   Select_Loop_Task.Receive_Message ("1");
   Select_Loop_Task.Send_Message ("A");
   Select_Loop_Task.Send_Message ("B");
   Select_Loop_Task.Receive_Message ("2");
   Select_Loop_Task.Stop;
exception
   when Tasking_Error =>
      Put_Line ("Expected exception: Entry not reached");
end Main;
```

Quiz

```
task T is
   entry E1;
   entry E2;
end T;
...
task body Other_Task is
begin
   select
      T.E1;
   or
      T.E2;
   end select;
end Other_Task;
```

What is the result of compiling and running this code?

A. `T.E1` is called
B. Nothing
C. Compilation error
D. Run-time 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. Run-time 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. Run-time 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. Run-time 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. Run-time 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. Run-time 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. Run-time 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. Run-time error

`then abort` aborts the select only, not `Main`.

Quiz

```
procedure Main is
    Ok : Boolean := False

    protected type O is
       entry P;
    end O;

    protected body O is
    begin
       entry P when Ok is
          Put_Line ("OK");
       end P;
    end O;

    Protected_Instance : O;

begin
    Protected_Instance.P;
end Main;
```

What is the result of compiling and running this code?

A. `OK = True`
B. Nothing
C. Compilation error
D. Run-time error

Quiz

```
procedure Main is
    Ok : Boolean := False

    protected type O is
       entry P;
    end O;

    protected body O is
    begin
       entry P when Ok is
          Put_Line ("OK");
       end P;
    end O;

    Protected_Instance : O;

begin
    Protected_Instance.P;
end Main;
```

What is the result of compiling and running this code?

A. `OK = True`
B. Nothing
C. Compilation error
D. Run-time 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

Tasking Control

Synchronous Task Control

- Primitives synchronization mechanisms and two-stage suspend operation
  - No critical section
  - More lightweight than protected objects
- Package exports a **Suspension_Object** type
  - Values are True and False, initially False
  - Such objects are awaited by (at most) one task
    - But can be set by several tasks

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);
   procedure Suspend_Until_True (S : in out Suspension_Object);
   function Current_State (S : Suspension_Object) return Boolean;
private
   ...
end Ada.Synchronous_Task_Control;
```

Timing Events

- User-defined actions executed at a specified wall-clock time
  - Calls back an **access protected procedure**
- Do not require a **task** or a **delay** statement

```
package Ada.Real_Time.Timing_Events is
   type Timing_Event is tagged limited private;
   type Timing_Event_Handler is
      access protected procedure
         (Event : in out Timing_Event);
   procedure Set_Handler
      (Event   : in out Timing_Event;
       At_Time : Time;
       Handler : Timing_Event_Handler);
   function Current_Handler
      (Event : Timing_Event)
       return Timing_Event_Handler;
   procedure Cancel_Handler
      (Event     : in out Timing_Event;
       Cancelled : out Boolean);
   function Time_Of_Event
      (Event : Timing_Event)
       return Time;
private
   ...
end Ada.Real_Time.Timing_Events;
```

Execution Time Clocks

- Not specific to Ravenscar / Jorvik
- Each task has an associated CPU time clock
  - Accessible via function call
- Clocks starts at creation time
  - **Before** activation
- Measures the task's total execution time
  - Including calls to libraries, O/S services...
  - But not including time in a blocked or suspended state
- System and runtime also execute code
  - As well as interrupt handlers
  - Their execution time clock assignment is implementation-defined

Partition Elaboration Control

- Library units are elaborated in a partially-defined order
  - They can declare tasks and interrupt handlers
  - Once elaborated, tasks start executing
  - Interrupts may occur as soon as hardware is enabled
    - May be during elaboration
- This can cause race conditions
  - Not acceptable for certification
- `pragma Partition_Elaboration_Policy`

Partition Elaboration Policy

- `pragma Partition_Elaboration_Policy`
  - Defined in RM Annex H "High Integrity Systems"
- Controls tasks' activation
- Controls interrupt attachment
- Always relative to library units' elaboration
- **Concurrent policy**
  - Activation at the end of declaration's scope elaboration
  - Ada default policy
- **Sequential policy**
  - Deferred activation and attachment until **all** library units are activated
  - Easier scheduling analysis

Lab

Tasking In Depth Lab

- Requirements
  - Create a datastore to set/inspect multiple "registers"
    - Individual registers can be read/written by multiple tasks
  - Create a "monitor" capability that will periodically update each register
    - Each register has it's own update frequency
  - Main program should print register values on request
- Hints
  - Datastore needs to control access to its contents
  - One task per register is easier than one task trying to maintain multiple update frequencies

Tasking In Depth Lab Solution - Datastore

```
package Datastore is
  type Register_T is (One, Two, Three);

  function Read (Register : Register_T) return Integer;
  procedure Write (Register : Register_T;
                   Value    : Integer);
end Datastore;

package body Datastore is
  type Register_Data_T is array (Register_T) of Integer;

  protected Registers is
    function Read (Register : Register_T) return Integer;
    procedure Write (Register : Register_T;
                     Value    : Integer);
  private
    Register_Data : Register_Data_T;
  end Registers;

  protected body Registers is
    function Read (Register : Register_T) return Integer is
       (Register_Data (Register));
    procedure Write (Register : Register_T;
                     Value    : Integer) is
    begin
      Register_Data (Register) := Value;
    end Write;
  end Registers;

  function Read (Register : Register_T) return Integer is
     (Registers.Read (Register));
  procedure Write (Register : Register_T;
                   Value    : Integer) is
  begin
    Registers.Write (Register, Value);
  end Write;
end Datastore;
```

Tasking In Depth Lab Solution - Monitor Task Type

```
with Datastore;
package Counter is
  task type Counter_T is
    entry Initialize (Register   : Datastore.Register_T;
                      Value      : Integer;
                      Increment  : Integer;
                      Delay_Time : Duration);
  end Counter_T;
end Counter;

package body Counter is
  task body Counter_T is
    O_Register  : Datastore.Register_T;
    O_Increment : Integer;
    O_Delay     : Duration;
    Initialized : Boolean := False;
  begin
    loop
      select
        accept Initialize (Register   : Datastore.Register_T;
                           Value      : Integer;
                           Increment  : Integer;
                           Delay_Time : Duration) do
          O_Register  := Register;
          O_Increment := Increment;
          O_Delay     := Delay_Time;
          Datastore.Write (Register => O_Register,
                           Value    => Value);
          Initialized := True;
        end Initialize;
      or
        delay O_Delay;
        if Initialized then
          Datastore.Write (Register => O_Register,
                           Value    => Datastore.Read (O_Register) + O_Increment);
        end if;
      end select;
    end loop;
  end Counter_T;
end Counter;
```

Tasking In Depth Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Counter;     use Counter;
with Datastore;   use Datastore;
procedure Main is
  Counters : array (Register_T) of Counter_T;

  function Get (Prompt : String) return Integer is
  begin
    Put ("  " & Prompt & ">");
    return Integer'Value (Get_Line);
  end Get;

  procedure Print is
  begin
    for Register in Register_T loop
      Put_Line (Register'Image & " =>" & Integer'Image (Datastore.Read (Register)));
    end loop;
  end Print;

begin
  for Register in Register_T loop
    Put_Line ("Register " & Register'Image);
    declare
      V : constant Integer := Get ("Initial value");
      I : constant Integer := Get ("Increment");
      D : constant Integer := Get ("Delay in tenths");
    begin
      Counters (Register).Initialize (Register   => Register,
                                      Value      => V,
                                      Increment  => I,
                                      Delay_Time => Duration (D) / 10.0);
    end;
  end loop;

  loop
    Put_Line ("Enter Q to quit, any other value to print registers");
    declare
      Str : constant String := Get_Line;
    begin
      exit when Str'Length > 0 and then (Str (Str'First) in 'Q' | 'q');
      Print;
    end;
  end loop;

  for Register in Register_T loop
    abort Counters (Register);
  end loop;
end Main;
```

Summary

Summary

- Tasks are **language-based** concurrency mechanisms
  - Typically implemented as threads
  - Not necessarily for **truly** parallel operations
  - Originally for task-switching / time-slicing
- Multiple mechanisms to **synchronize** tasks
  - Delay
  - Rendezvous
  - Queues
  - Protected Objects

Ada Contracts

Introduction

Design-By-Contract

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - *Contract* specifies *requirements* or *guarantees*
    - *"A specification of a software element that affects its use by potential clients."* (Bertrand Meyer)
  - *Supplier* provides services
    - Guarantees specific functional behavior
    - Has requirements for guarantees to hold
  - *Client* utilizes services
    - Guarantees supplier's conditions are met
    - Requires result to follow the subprogram's guarantees

Ada Contracts

- Ada contracts include enforcement
  - At compile-time: specific constructs, features, and rules
  - At run-time: language-defined and user-defined exceptions
- Facilities as part of the language definition
  - Range specifications
  - Parameter modes
  - Generic contracts
  - OOP `interface` types
  - Work well, but on a restricted set of use-cases
- Contract aspects to be more expressive
  - Carried by subprograms
  - ... or by types (seen later)
  - Can have **arbitrary** conditions, more **versatile**

Assertion

- Boolean expression expected to be True
- Said *to hold* when True
- Language-defined **pragma**
  - The Ada.Assertions.Assert subprogram can wrap it

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

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

```
package Ada.Assertions is
  Assertion_Error : exception;
  procedure Assert (Check : in Boolean);
  procedure Assert (Check : in Boolean; Message : in String);
end Ada.Assertions;
```

Defensive Programming

- Should be replaced by subprogram contracts when possible

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

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

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

   Acquire (P);
end;
```

Quiz

Which of the following statements is (are) correct?

A. Contract principles apply only to newer versions of the language
B. Contract should hold even for unique conditions and corner cases
C. Contract principles were first implemented in Ada
D. You cannot be both supplier and client

Quiz

Which of the following statements is (are) correct?

A. Contract principles apply only to newer versions of the language
B. ***Contract should hold even for unique conditions and corner cases***
C. Contract principles were first implemented in Ada
D. You cannot be both supplier and client

Explanations

A. No, but design-by-contract **aspects** were fully integrated into Ada 2012
B. Yes, special case should be included in the contract
C. No, in eiffel, in 1986!
D. No, in fact you are always **both**, even the `Main` has a caller!

Quiz

Which of the following statements is (are) correct?

A. Assertions can be used in declarations
B. Assertions can be used in expressions
C. Any corrective action should happen before contract checks
D. Assertions must be checked using `pragma Assert`

Quiz

Which of the following statements is (are) correct?

A. ***Assertions can be used in declarations***
B. Assertions can be used in expressions
C. ***Any corrective action should happen before contract checks***
D. Assertions must be checked using `pragma Assert`

Explanations

A. Will be checked at elaboration
B. No assertion expression, but `raise` expression exists
C. Exceptions as flow-control adds complexity, prefer a proactive `if` to a (reactive) `exception` handler
D. You can call `Ada.Assertions.Assert`, or even directly `raise Assertion_Error`

Quiz

Which of the following statements is (are) correct?

A. Defensive coding is a good practice
B. Contracts can replace all defensive code
C. Contracts are executable constructs
D. Having exhaustive contracts will prevent runtime errors

Quiz

Which of the following statements is (are) correct?

A. ***Defensive coding is a good practice***
B. Contracts can replace all defensive code
C. Contracts are executable constructs
D. Having exhaustive contracts will prevent runtime errors

Explanations

A. Principles are sane, contracts extend those
B. See previous slide example
C. e.g. generic contracts are resolved at compile-time
D. A failing contract **will cause** a runtime error, only extensive (dynamic / static) analysis of contracted code may provide confidence in the absence of runtime errors (AoRTE)

Preconditions and Postconditions

Subprogram-based Assertions

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

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

Requirements / Guarantees: Quiz

- Given the following piece of code

```
procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
 with Pre => Has_Power,
      Post => Is_On;
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)		
Post (Is_On)		

Requirements / Guarantees: Quiz

- Given the following piece of code

```
procedure Start is
begin
    ...
    Turn_On;
    ...

procedure Turn_On
 with Pre => Has_Power,
      Post => Is_On;
```

- Complete the table in terms of requirements and guarantees

	Client (Start)	Supplier (Turn_On)
Pre (Has_Power)	Requirement	Guarantee
Post (Is_On)	Guarantee	Requirement

Examples

```
package Stack_Pkg is
   procedure Push (Item : in Integer) with
         Pre  => not Full,
         Post => not Empty and then Top = Item;
   procedure Pop (Item : out Integer) with
         Pre  => not Empty,
         Post => not Full;
   function Pop return Integer with
         Pre  => not Empty,
         Post => not Full;
   function Top return Integer with
         Pre => not Empty;
   function Empty return Boolean;
   function Full return Boolean;
end Stack_Pkg;

package body Stack_Pkg is
   Values  : array (1 .. 100) of Integer;
   Current : Natural := 0;

   -- Push/Pop cannot fail because preconditions prevent it
   procedure Push (Item : in Integer) is
   begin
      Current          := Current + 1;
      Values (Current) := Item;
   end Push;

   procedure Pop (Item : out Integer) is
   begin
      Item    := Values (Current);
      Current := Current - 1;
   end Pop;

   function Pop return Integer is
      Item : constant Integer := Values (Current);
   begin
      Current := Current - 1;
      return Item;
   end Pop;

   function Top return Integer is (Values (Current));
   function Empty return Boolean is (Current not in Values'Range);
   function Full return Boolean is (Current >= Values'Length);
end Stack_Pkg;
```

Preconditions

- Define obligations on client for successful call
  - Precondition specifies required conditions
  - Clients must meet precondition for supplier to succeed
- Boolean expressions
  - Arbitrary complexity
  - Specified via aspect name `Pre`
- Checked prior to call by client
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre => not Full (This);
```

Postconditions

- Define obligations on supplier
  - Specify guaranteed conditions after call
- Boolean expressions (same as preconditions)
  - Specified via aspect name `Post`
- Content as for preconditions, plus some extras
- Checked after corresponding subprogram call
  - `Assertion_Error` raised if false

```
procedure Push (This : in out Stack;  Value : Content)
  with Pre  => not Full (This),
       Post => not Empty (This) and Top (This) = Value;
...
function Top (This : Stack) return Content
  with Pre => not Empty (This);
```

Postcondition 'Old Attribute

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

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

Function Postcondition `'Result` Attribute

- **`function`** result can be manipulated with `'Result`

Preconditions and Postconditions Example

- Multiple aspects separated by commas

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

Quiz

```
function Area (L : Positive; H : Positive) return Positive is
   (L * H)
with Pre => ?
```

Which pre-condition is necessary for `Area` to calculate the correct result for all values `L` and `H`?

A. `L > 0 and H > 0`
B. `L < Positive'Last and H < Positive'Last`
C. `L * H in Positive`
D. None of the above

Quiz

```
function Area (L : Positive; H : Positive) return Positive is
   (L * H)
with Pre => ?
```

Which pre-condition is necessary for Area to calculate the correct result for all values L and H?

A. `L > 0 and H > 0`
B. `L < Positive'Last and H < Positive'Last`
C. `L * H in Positive`
D. ***None of the above***

Explanations

A. Parameters are `Positive`, so this is unnecessary
B. Overflow for large numbers
C. Classic trap: the check itself may cause an overflow!

The correct precondition would be

`Integer'Last / L <= H`

to prevent overflow errors on the range check.

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for component at I = I
Database : array (Index_T) of Integer;
-- Set the value for component Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move (-1, 10)`

```
Database'Old (Index)
Database (Index`Old)
Database (Index)'Old
```

Quiz

```
type Index_T is range 1 .. 100;
-- Database initialized such that value for component at I = I
Database : array (Index_T) of Integer;
-- Set the value for component Index to Value and
-- then increment Index by 1
function Set_And_Move (Value :        Integer;
                       Index : in out Index_T)
                       return Boolean
   with Post => ...
```

Given the following expressions, what is their value if they are evaluated in the postcondition of the call `Set_And_Move (-1, 10)`

`Database'Old (Index)`	11	Use new index in copy of original Database
``Database (Index`Old)``	-1	Use copy of original index in current Database
`Database (Index)'Old`	10	Evaluation of Database (Index) before call

Separations of Concerns

- Pre and Post fit together

```
function Val return Integer
 with Post => F'Result /= 0
is (if Val_Raw > 0 then Val_Raw else 1);

procedure Process (I : Integer)
 with Pre => I /= 0
is (Set_Output (10 / I));

[...]

Process (Val);
```

- Review of interface: guaranteed to work
  - What is returned by `Val` is always valid for `Process`
  - Need to check implementations
- Review of implementation
  - `Val` **always** returns a value that is /= 0
  - `Process` accepts **any** value that is /= 0
- Great separation of concerns
  - a team (Clients) could be in charge of reviewing the interface part
  - another team (Suppliers) could be in charge of reviewing the implementation part
  - both would use the contracts as a common understanding
  - Tools can do an automated review / validation: GNAT Static Analysis Suite, SPARK

No Secret Precondition Requirements

- Client should be able to **guarantee** them
- Enforced by the compiler

```
package P is
  function Foo return Bar
    with Pre => Hidden; -- illegal private reference
private
  function Hidden return Boolean;
end P;
```

Postconditions Are Good Documentation

```
procedure Reset
    (Unit : in out DMA_Controller;
     Stream : DMA_Stream_Selector)
  with Post =>
    not Enabled (Unit, Stream) and
    Operating_Mode (Unit, Stream) = Normal_Mode and
    Selected_Channel (Unit, Stream) = Channel_0 and
    not Double_Buffered (Unit, Stream) and
    Priority (Unit, Stream) = Priority_Low and
    (for all Interrupt in DMA_Interrupt =>
        not Interrupt_Enabled (Unit, Stream, Interrupt));
```

Contracts Code Reuse

- Contracts are about **usage** and **behaviour**
  - Not optimization
  - Not implementation details
  - **Abstraction** level is typically high
- Extracting them to `function` is a good idea
  - *Code as documentation, executable specification*
  - Completes the **interface** that the client has access to
  - Allows for **code reuse**

```
procedure Withdraw (This   : in out Account;
                    Amount :        Currency) with
  Pre  => Open (This) and Funds_Available (This, Amount),
  Post => Balance (This) = Balance (This)'Old - Amount;
...
function Funds_Available (This   : Account;
                          Amount : Currency)
                          return Boolean is
    (Amount > 0.0 and then Balance (This) >= Amount)
  with Pre => Open (This);
```

- A `function` may be unavoidable
  - Referencing private type components

Assertion Policy

- Assertions checks can be controlled with `pragma Assertion_Policy`

```
pragma Assertion_Policy
     (Pre => Check,
      Post => Ignore);
```

- Fine **granularity** over assertion kinds and policy identifiers

https://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/implementation_defined_pragmas.html#pragma-assertion-policy

- Certain advantage over explicit checks which are **harder** to disable
  - Conditional compilation via global `constant Boolean`

```
procedure Push (This : in out Stack;  Value : Content) is
begin
  if Debugging then
    if Full (This) then
      raise Overflow;
    end if;
  end if;
```

Type Invariants

Strong Typing

- Ada supports strong typing

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

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

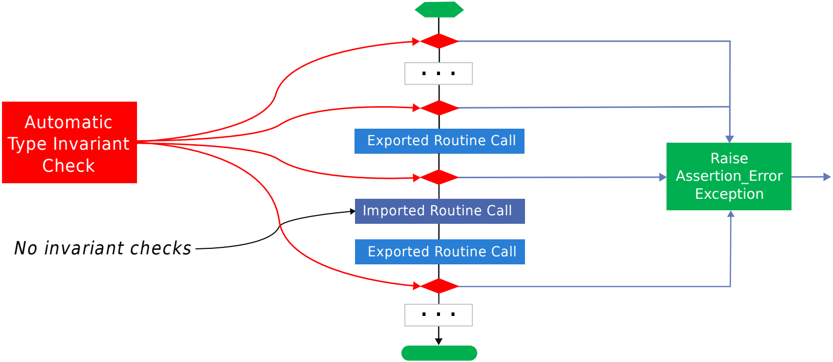
Examples

```
package Bank is
   type Account_T is private with Type_Invariant => Consistent_Balance (Account_T);
   type Currency_T is delta 0.01 digits 12;
   function Consistent_Balance (This : Account_T) return Boolean;
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T);
private
   type Vector_T is array (1 .. 100) of Currency_T;
   type Transaction_Vector_T is record
      Values : Vector_T;
      Count  : Natural := 0;
   end record;
   type Account_T is record -- initial state MUST satisfy invariant
      Current_Balance : Currency_T := 0.0;
      Withdrawals     : Transaction_Vector_T;
      Deposits        : Transaction_Vector_T;
   end record;
end Bank;

package body Bank is
   function Total (This : Transaction_Vector_T) return Currency_T is
      Result : Currency_T := 0.0;
   begin
      for I in 1 .. This.Count loop -- no iteration if list empty
         Result := Result + This.Values (I);
      end loop;
      return Result;
   end Total;
   function Consistent_Balance (This : Account_T) return Boolean is
      (Total (This.Deposits) - Total (This.Withdrawals) = This.Current_Balance);
   procedure Open (This : in out Account_T; Initial_Deposit : Currency_T) is
   begin
      This.Current_Balance := Initial_Deposit;
      -- if we checked, the invariant would be false here!
      This.Withdrawals.Count   := 0;
      This.Deposits.Count      := 1;
      This.Deposits.Values (1) := Initial_Deposit;
   end Open; -- invariant is now true
end Bank;
```

Type Invariant

- Applied to `private` types
- Evaluated as postcondition of creation, evaluation, or return object
  - When objects first created
  - Assignment by clients
  - Type conversions
    - Creates new instances
- Not evaluated on internal state changes
  - Internal routine calls
  - Internal assignments
- Remember - these are abstract data types

Invariant Over Object Lifetime (Calls)



Example Type Invariant

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

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

Invariants Don't Apply Internally

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

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

Quiz

```
package P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      (True);
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1
B. 2
C. 3
D. 4

Quiz

```
package P is
   type Some_T is private;
   procedure Do_Something (X : in out Some_T);
private
   function Counter (I : Integer) return Boolean;
   type Some_T is new Integer with
      Type_Invariant => Counter (Integer (Some_T));
end P;

package body P is
   function Local_Do_Something (X : Some_T)
                                return Some_T is
      Z : Some_T := X + 1;
   begin
      return Z;
   end Local_Do_Something;
   procedure Do_Something (X : in out Some_T) is
   begin
      X := X + 1;
      X := Local_Do_Something (X);
   end Do_Something;
   function Counter (I : Integer)
                     return Boolean is
      (True);
end P;
```

If **Do_Something** is called from outside of P, how many times is **Counter** called?

A. 1
B. *2*
C. 3
D. 4

Type Invariants are only evaluated on entry into and exit from externally visible subprograms. So `Counter` is called when entering and exiting `Do_Something` - not `Local_Do_Something`, even though a new instance of `Some_T` is created

Subtype Predicates

Examples

```ada
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
procedure Predicates is

   subtype Even_T is Integer with Dynamic_Predicate => Even_T mod 2 = 0;
   type Serial_Baud_Rate_T is range 110 .. 115_200 with
      Static_Predicate => Serial_Baud_Rate_T in -- Non-contiguous range
          2_400 | 4_800 | 9_600 | 14_400 | 19_200 | 28_800 | 38_400 | 56_000;

   -- This must be dynamic because "others" will be evaluated at run-time
   subtype Vowel_T is Character with Dynamic_Predicate =>
         (case Vowel_T is when 'A' | 'E' | 'I' | 'O' | 'U' => True, when others => False);

   type Table_T is array (Integer range <>) of Integer;
   subtype Sorted_Table_T is Table_T (1 .. 5) with
        Dynamic_Predicate =>
        (for all K in Sorted_Table_T'Range =>
           (K = Sorted_Table_T'First or else Sorted_Table_T (K - 1) <= Sorted_Table_T (K)));

   J      : Even_T;
   Values : Sorted_Table_T := (1, 3, 5, 7, 9);

begin
   begin
      Put_Line ("J is" & J'Image);
      J := Integer'Succ (J); -- assertion failure here
      Put_Line ("J is" & J'Image);
      J := Integer'Succ (J); -- or maybe here
      Put_Line ("J is" & J'Image);
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;

   for Baud in Serial_Baud_Rate_T loop
      Put_Line (Baud'Image);
   end loop;

   Put_Line (Vowel_T'Image (Vowel_T'Succ ('A')));
   Put_Line (Vowel_T'Image (Vowel_T'Pred ('Z')));

   begin
      Values (3) := 0; -- not an exception
      Values     := (1, 3, 0, 7, 9); -- exception
   exception
      when The_Err : others =>
         Put_Line (Exception_Message (The_Err));
   end;
end Predicates;
```

Predicates

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

Why Two Predicate Forms?

	Static	Dynamic
Content	More Restricted	Less Restricted
Placement	Less Restricted	More Restricted

- Static predicates can be used in more contexts
  - More restrictions on content
  - Can be used in places Dynamic Predicates cannot
- Dynamic predicates have more expressive power
  - Fewer restrictions on content
  - Not as widely available

Subtype Predicate Examples

- Dynamic Predicate

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

- Static Predicate

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

Predicate Checking

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

Predicate Expression Content

- Reference to value of type itself, i.e., "current instance"

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
J, K : Even := 42;
```

- Any visible object or function in scope
  - Does not have to be defined before use
  - Relaxation of "declared before referenced" rule of linear elaboration
  - Intended especially for (expression) functions declared in same package spec

Static Predicates

- *Static* means known at compile-time, informally
  - Language defines meaning formally (RM 3.2.4)
- Allowed in contexts in which compiler must be able to verify properties
- Content restrictions on predicate are necessary
- Ordinary Ada static expressions
- Static membership test selected by current instance
- Example

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

Dynamic Predicate Expression Content

- Any arbitrary boolean expression
  - Hence all allowed static predicates' content
- Plus additional operators, etc.

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;
subtype Vowel is Character with Dynamic_Predicate =>
  (case Vowel is
    when 'A' | 'E' | 'I' | 'O' | 'U' => True,
    when others => False); -- evaluated at run-time
```

- Plus calls to functions
  - User-defined
  - Language-defined

Beware Accidental Recursion in Predicate

- Involves functions because predicates are expressions
- Caused by checks on function arguments
- Infinitely recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
   Dynamic_Predicate => Sorted (Sorted_Table);
-- on call, predicate is checked!
function Sorted (T : Sorted_Table) return Boolean;
```

- Non-recursive example

```
type Sorted_Table is array (1 .. N) of Integer with
   Dynamic_Predicate =>
   (for all K in Sorted_Table'Range =>
      (K = Sorted_Table'First
       or else Sorted_Table (K - 1) <= Sorted_Table (K)));
```

- Type-based example

```
type Table is array (1 .. N) of Integer;
subtype Sorted_Table is Table with
     Dynamic_Predicate => Sorted (Sorted_Table);
function Sorted (T : Table) return Boolean;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A.
```
subtype T is Days_T with
   Static_Predicate => T in Sun | Sat;
```

B.
```
subtype T is Days_T with Static_Predicate =>
   (if T = Sun or else T = Sat then True else False);
```

C.
```
subtype T is Days_T with
   Static_Predicate => not Is_Weekday (T);
```

D.
```
subtype T is Days_T with
   Static_Predicate =>
     case T is when Sat | Sun => True,
               when others => False;
```

Quiz

```
type Days_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
function Is_Weekday (D : Days_T) return Boolean is
   (D /= Sun and then D /= Sat);
```

Which of the following is a valid subtype predicate?

A. *`subtype T is Days_T with`*
   `Static_Predicate => T in Sun | Sat;`

B. ```
   subtype T is Days_T with Static_Predicate =>
      (if T = Sun or else T = Sat then True else False);
   ```

C. ```
   subtype T is Days_T with
      Static_Predicate => not Is_Weekday (T);
   ```

D. ```
   subtype T is Days_T with
      Static_Predicate =>
        case T is when Sat | Sun => True,
                  when others => False;
   ```

Explanations

A. Correct
B. `If` statement not allowed in a predicate
C. Function call not allowed in `Static_Predicate` (this would be OK for `Dynamic_Predicate`)
D. Missing parentheses around `case` expression
   ```

Summary

Working with Type Invariants

- They are not completely foolproof
  - External corruption is possible
  - Requires dubious usage
- Violations are intended to be supplier bugs
  - But not necessarily so, since not always bullet-proof
- However, reasonable designs will be foolproof

Type Invariants Vs Predicates

- Type Invariants are valid at external boundary
    - Useful for complex types - type may not be consistent during an operation
- Predicates are like other constraint checks
    - Checked on declaration, assignment, calls, etc

Contract-Based Programming Benefits

- Facilitates building software with reliability built-in
  - Software cannot work well unless "well" is carefully defined
  - Clarifies design by defining obligations/benefits
- Enhances readability and understandability
  - Specification contains explicitly expressed properties of code
- Improves testability but also likelihood of passing!
- Aids in debugging
- Facilitates tool-based analysis
  - Compiler checks conformance to obligations
  - Static analyzers (e.g., SPARK, GNAT Static Analysis Suite) can verify explicit preconditions and postconditions