# Day 1 - AM

# 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**

# 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

# 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

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

Which statement(s) is (are) legal?

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

# Quiz

```
with Interfaces; use Interfaces;

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

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

Which statement(s) is (are) legal?

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

# Quiz

```
with Interfaces; use Interfaces;

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

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

Which statement(s) is (are) legal?

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

# Representation Values

# Enumeration Representation Values

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

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

# 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

# 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

# Day 1 - PM

# 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

# Private Types

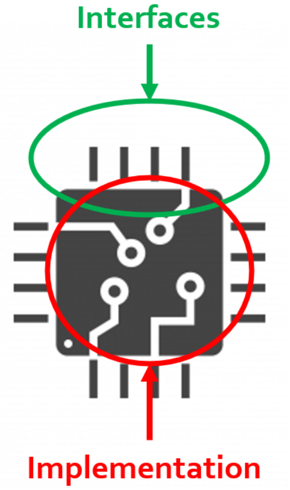
# Introduction

# 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

```ada
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

# 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

```
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;
```

# 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. *(correct)*
```
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. *(correct)*
```
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

# Day 2 - AM

# Advanced Data Hiding

# Type Views

# Capabilities / Constraints of a Type

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

# Partial Vs Full View of a Type

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

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

# Discriminants

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

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

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

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

# Unknown Constraint

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

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

# Limited

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

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

# Tagged

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

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

# Private Primitives

- Primitives can be either public or private
- Privacy is **orthogonal** with type hierarchy
  - Derived types **may not** have access to private primitives
  - Child packages **can** access private part
    - and call the private primitive directly
- A primitive that has to be derived **must** be public
  - Abstract, constructor...

```
package P is
   type T is private;
   -- abstract must be public
   procedure Execute (Obj : T) is abstract;
   -- constructor must be public
   function Make return T;
private
   procedure Internal_Reset (Obj : T); -- can be private
end package P;
```

# Tagged Extension

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

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

# Tagged Abstract

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

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

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

# Protection Idiom

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

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

- Helps keeping track of the object usage

# Quiz

```
type T is private;
```

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

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

# Quiz

```
type T is private;
```

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

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

A. Can declare supplementary capability
B. Cannot add further constraint
C. Note: an unconstrained **range** <> would be incorrect
D. Abstract is a constraint

# Incomplete Types

# Incomplete Types

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

# How to Get an Incomplete Type View?

- From an explicit declaration

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

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

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

# Type Completion Deferred to the Body

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

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

# Quiz

```
type T;
```

In the same scope, which of the following types is (are) legal?

A. `type Acc is access T`

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

C. `type T2 is new T`

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

# Quiz

```
type T;
```

In the same scope, which of the following types is (are) legal?

A. *`type Acc is access T`*

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

C. `type T2 is new T`

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

A. Can `access` the type
B. Cannot use the type as a component
C. Cannot derive from an incomplete type
D. Be careful about the use of an anonymous type here!

# Quiz

```ada
package Pkg is
   type T is private;
private
   -- Declarations Here
```

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

A. `type T is array (Positive range <>) of Integer`

B. `type T is tagged null record`

C. `type T is limited null record`

D. ```ada
   type T_Arr is array (Positive range <>) of T;
   type T is new Integer;
   ```
   ```

# Quiz

```
package Pkg is
   type T is private;
private
   -- Declarations Here
```

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

A. `type T is array (Positive range <>) of Integer`

B. *`type T is tagged null record`*

C. `type T is limited null record`

D. *`type T_Arr is array (Positive range <>) of T;`*
   *`type T is new Integer;`*

A. Cannot complete with an unconstrained type
B. Can complete with the `tagged` capability
C. Cannot complete with a `limited` constraint
D. Even though T is private, it can be used as component

# Private Library Units

# Child Units and Privacy

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

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

```
package Root.Child is
  X1 : T; -- illegal
private
  X2 : T;
end Root.Child;
```

- *Private child* can view the private part
  - Used for "implementation details"

# Importing a Private Child

- A **private package** can view its **parent** **private** part
- A private package's usage (*view*) is
  - Restricted to the *Private descendents of their parent*
  - Visible from parent's **body**
  - Visible from public sibling's **private** section, and **body**
  - Visible from private siblings (public, **private**, **body**)

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

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

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

# Private Children and **with**

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

*Public package spec cannot* **with** *a private package*

```
with Root.Private_Child;
package Root.Bad_Child is
   Object1 : Root.Private_Child.T;
   procedure Proc2;
private
   Object2 : Root.Private_Child.T;
end Root.Bad_Child;
```

```
root-bad_child.ads:1:06: error:
current unit must also be private
descendant of "Root"
```

*But it can* **with** *a sibling private package from its body*

```
package Root.Good_Child is
   procedure Proc2;
end Root.Good_Child;

with Root.Private_Child;
package body Root.Good_Child is
   Object1 : Root.Private_Child.T;
   Object2 : Root.Private_Child.T;
   procedure Proc2 is null;
end Root.Good_Child;
```

# `private with`

- The parent and its children can `private with` a private package
  - From anywhere
  - View given **stays** `private`

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

- Clients of `Root.Child2` don't have any visibility on `Root.Child1`

# Children "Inherit" From Private Properties of Parent

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

```
package Root is
end Root;

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

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

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

# Lab

# Advanced Data Hiding Lab

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

# Advanced Data Hiding Lab Solution - Message Type

```
package Messages is
   type Message_T is private;

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

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

package body Messages is
   type Message_Content_T is new Integer;

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

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

# Advanced Data Hiding Lab Solution - Message List Type

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

# Advanced Data Hiding Lab Solution - Message List Operations

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

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

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

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

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

# Advanced Data Hiding Lab Solution - Main

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

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

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

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

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

# Summary

# Summary

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

# 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**

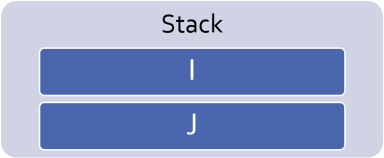
- Tightly coupled to dynamically allocated objects
- Used with Ada's controlled memory management (pools)
- Can only point to object created by `new`
- Memory management tied to specific storage pool

# Access Types Can Be Dangerous

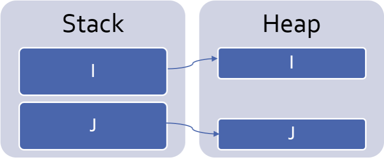
- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot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

# Day 2 - PM

# 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

```ada
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

```ada
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

# Day 3 - AM

# 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

```ada
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

```
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

```
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

# Interfacing with C

# Introduction

# Introduction

- Lots of C code out there already
    - Maybe even a lot of reusable code in your own repositories
- Need a way to interface Ada code with existing C libraries
    - Built-in mechanism to define ability to import objects from C or export Ada objects
- Passing data between languages can cause issues
    - Sizing requirements
    - Passing mechanisms (by reference, by copy)

# Import / Export

# Import / Export Aspects (1/2)

- Aspects `Import` and `Export` allow Ada and C to interact
  - `Import` indicates a subprogram imported into Ada
  - `Export` indicates a subprogram exported from Ada
- Need aspects definining calling convention and external name
  - `Convention => C` tells linker to use C-style calling convention
  - `External_Name => "<name>"` defines object name for linker
- Ada implementation

```
procedure Imported_From_C with
   Import,
   Convention    => C,
   External_Name => "SomeProcedureInC";

procedure Exported_To_C with
   Export,
   Convention    => C,
   External_Name => "some_ada_procedure;
```

- C implementation

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

extern void ada_some_procedure (void);
```

# Import / Export Aspects (2/2)

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

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

  - C implementation

```
int my_var;
```

# Import / Export with Pragmas

- You can also use **pragma** to import/export entities

```
procedure C_Some_Procedure;
pragma Import (C, C_Some_Procedure, "SomeProcedure");

procedure Some_Procedure;
pragma Export (C, Some_Procedure, "ada_some_procedure");
```

# Parameter Passing

# Parameter Passing to/from C

- The mechanism used to pass formal subprogram parameters and function results depends on:
    - The type of the parameter
    - The mode of the parameter
    - The Convention applied on the Ada side of the subprogram declaration
- The exact meaning of *Convention C*, for example, is documented in *LRM* B.1 - B.3, and in the *GNAT User's Guide* section 3.11.

# Passing Scalar Data As Parameters

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

```
with Interfaces.C;
function C_Proc (I : Interfaces.C.Int)
    return Interfaces.C.Int;
pragma Import (C, C_Proc, "c_proc");
```

- C view

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

# Passing Structures As Parameters

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

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

- Ada View

- This can also be done with pragmas

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

# Parameter Modes

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

```
Type R1 is record
   V : int;
end record
with Convention => C;

type R2 is record
   V : int;
end record
with Convention => C_Pass_By_Copy;
```

- C View

```
struct R1{
   int V;
};
struct R2 {
   int V;
};
void f1 (R1 p);
void f2 (R2 p);
```

# Complex Data Types

# Unions

- C `union`

```
union Rec {
   int A;
   float B;
};
```

- C unions can be bound using the `Unchecked_Union` aspect
- These types must have a mutable discriminant for convention purpose, which doesn't exist at run-time
  - All checks based on its value are removed - safety loss
  - It cannot be manually accessed
- Ada implementation of a C `union`

```
type Rec (Flag : Boolean := False) is
record
   case Flag is
      when True =>
         A : int;
      when False =>
         B : float;
   end case;
end record
with Unchecked_Union,
     Convention => C;
```

# Arrays Interfacing

- In Ada, arrays are of two kinds:
  - Constrained arrays
  - Unconstrained arrays
- Unconstrained arrays are associated with
  - Components
  - Bounds
- In C, an array is just a memory location pointing (hopefully) to a structured memory location
  - C does not have the notion of unconstrained arrays
- Bounds must be managed manually
  - By convention (null at the end of string)
  - By storing them on the side
- Only Ada constrained arrays can be interfaced with C

# Arrays From Ada to C

- An Ada array is a composite data structure containing 2 parts: Bounds and Components
  - **Fat pointers**
- When arrays can be sent from Ada to C, C will only receive an access to the components of the array
- Ada View

```
type Arr is array (Integer range <>) of int;
procedure P (V : Arr; Size : int);
pragma Import (C, P, "p");
```

- C View

```
void p (int * v, int size)  {
}
```

# Arrays From C to Ada

- There are no boundaries to C types, the only Ada arrays that can be bound must have static bounds
- Additional information will probably need to be passed
- Ada View

```
-- DO NOT DECLARE OBJECTS OF THIS TYPE
type Arr is array (0 .. Integer'Last) of int;

procedure P (V : Arr; Size : int);
pragma Export (C, P, "p");

procedure P (V : Arr; Size : int) is
begin
   for J in 0 .. Size - 1 loop
      -- code;
   end loop;
end P;
```

- C View

```
extern void p (int * v, int size);
int x [100];
p (x, 100);
```

# Strings

- Importing a `String` from C is like importing an array - has to be done through a constrained array
- `Interfaces.C.Strings` gives a standard way of doing that
- Unfortunately, C strings have to end by a null character
- Exporting an Ada string to C needs a copy!

```
Ada_Str : String := "Hello World";
C_Str : chars_ptr := New_String (Ada_Str);
```

- Alternatively, a knowledgeable Ada programmer can manually create Ada strings with correct ending and manage them directly

```
Ada_Str : String := "Hello World" & ASCII.NUL;
```

- Back to the unsafe world - it really has to be worth it speed-wise!

# Interfaces.C

# Interfaces.C Hierarchy

- Ada supplies a subsystem to deal with Ada/C interactions
- `Interfaces.C` - contains typical C types and constants, plus some simple Ada string to/from C character array conversion routines
  - `Interfaces.C.Extensions` - some additional C/C++ types
  - `Interfaces.C.Pointers` - generic package to simulate C pointers (pointer as an unconstrained array, pointer arithmetic, etc)
  - `Interfaces.C.Strings` - types / functions to deal with C `"char *"`

# Interfaces.C

```
package Interfaces.C is

   --  Declaration's based on C's <limits.h>
   CHAR_BIT  : constant := 8;
   SCHAR_MIN : constant := -128;
   SCHAR_MAX : constant := 127;
   UCHAR_MAX : constant := 255;

   type int   is new Integer;
   type short is new Short_Integer;
   type long  is range -(2 ** (System.Parameters.long_bits - Integer'(1)))
     .. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;

   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;

   type unsigned       is mod 2 ** int'Size;
   type unsigned_short is mod 2 ** short'Size;
   type unsigned_long  is mod 2 ** long'Size;

   type unsigned_char is mod (UCHAR_MAX + 1);
   for unsigned_char'Size use CHAR_BIT;

   type ptrdiff_t is range -(2 ** (System.Parameters.ptr_bits - Integer'(1))) ..
                           +(2 ** (System.Parameters.ptr_bits - Integer'(1)) - 1);

   type size_t is mod 2 ** System.Parameters.ptr_bits;

   --  Floating-Point
   type C_float     is new Float;
   type double      is new Standard.Long_Float;
   type long_double is new Standard.Long_Long_Float;

   type char is new Character;
   nul : constant char := char'First;

   function To_C   (Item : Character) return char;
   function To_Ada (Item : char)      return Character;

   type char_array is array (size_t range <>) of aliased char;
   for char_array'Component_Size use CHAR_BIT;

   function Is_Nul_Terminated (Item : char_array) return Boolean;

   -- (more not specified here)

end Interfaces.C;
```

# Interfaces.C.Extensions

```
package Interfaces.C.Extensions is

   --  Definitions for C "void" and "void *" types
   subtype void     is System.Address;
   subtype void_ptr is System.Address;

   --  Definitions for C incomplete/unknown structs
   subtype opaque_structure_def is System.Address;
   type opaque_structure_def_ptr is access opaque_structure_def;

   --  Definitions for C++ incomplete/unknown classes
   subtype incomplete_class_def is System.Address;
   type incomplete_class_def_ptr is access incomplete_class_def;

   --  C bool
   type bool is new Boolean;
   pragma Convention (C, bool);

   --  64-bit integer types
   subtype long_long is Long_Long_Integer;
   type unsigned_long_long is mod 2 ** 64;

   -- (more not specified here)

end Interfaces.C.Extensions;
```

# Interfaces.C.Pointers

```
generic
   type Index is (<>);
   type Component is private;
   type Component_Array is array (Index range <>) of aliased Component;
   Default_Terminator : Component;

package Interfaces.C.Pointers is

   type Pointer is access all Component;
   for Pointer'Size use System.Parameters.ptr_bits;

   function Value (Ref        : Pointer;
                   Terminator : Component := Default_Terminator)
                   return Component_Array;
   function Value (Ref    : Pointer;
                   Length : ptrdiff_t)
                   return Component_Array;

   Pointer_Error : exception;

   function "+" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
   function "+" (Left : ptrdiff_t; Right : Pointer)   return Pointer;
   function "-" (Left : Pointer;   Right : ptrdiff_t) return Pointer;
   function "-" (Left : Pointer;   Right : Pointer)   return ptrdiff_t;

   procedure Increment (Ref : in out Pointer);
   procedure Decrement (Ref : in out Pointer);

   -- (more not specified here)

end Interfaces.C.Pointers;
```

# Interfaces.C.Strings

```
package Interfaces.C.Strings is

   type char_array_access is access all char_array;
   for char_array_access'Size use System.Parameters.ptr_bits;

   type chars_ptr is private;

   type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;

   Null_Ptr : constant chars_ptr;

   function To_Chars_Ptr (Item      : char_array_access;
                          Nul_Check : Boolean := False) return chars_ptr;

   function New_Char_Array (Chars : char_array) return chars_ptr;

   function New_String (Str : String) return chars_ptr;

   procedure Free (Item : in out chars_ptr);

   function Value (Item : chars_ptr) return char_array;
   function Value (Item   : chars_ptr;
                   Length : size_t)
                   return char_array;
   function Value (Item : chars_ptr) return String;
   function Value (Item   : chars_ptr;
                   Length : size_t)
                   return String;

   function Strlen (Item : chars_ptr) return size_t;

   -- (more not specified here)

end Interfaces.C.Strings;
```

# Lab

# Interfacing with C Lab

- Requirements
  - Given a C function that calculates speed in MPH from some information, your application should
    - Ask user for distance and time
    - Populate the structure appropriately
    - Call C function to return speed
    - Print speed to console
- Hints
  - Structure contains the following components
    - Distance (floating point)
    - Distance Type (enumeral)
    - Seconds (floating point)

# Interfacing with C Lab - GNAT Studio

To compile/link the C file into the Ada executable:

1. Make sure the C file is in the same directory as the Ada source files
2. Edit → Project Properties
3. Sources → Languages → Check the "C" box
4. Build and execute as normal

# Interfacing with C Lab Solution - Ada

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

   package Float_Io is new Ada.Text_IO.Float_IO (Interfaces.C.C_float);

   One_Minute_In_Seconds : constant := 60.0;
   One_Hour_In_Seconds   : constant := 60.0 * One_Minute_In_Seconds;

   type Distance_T is (Feet, Meters, Miles) with Convention => C;
   type Data_T is record
      Distance      : Interfaces.C.C_float;
      Distance_Type : Distance_T;
      Seconds       : Interfaces.C.C_float;
   end record with Convention => C;
   function C_Miles_Per_Hour (Data : Data_T) return Interfaces.C.C_float
      with Import, Convention => C, External_Name => "miles_per_hour";

   Object_Feet : constant Data_T :=
     (Distance => 6_000.0,
      Distance_Type => Feet,
      Seconds  => One_Minute_In_Seconds);
   Object_Meters : constant Data_T :=
     (Distance => 3_000.0,
      Distance_Type => Meters,
      Seconds  => One_Hour_In_Seconds);
   Object_Miles : constant Data_T :=
     (Distance => 1.0,
      Distance_Type =>
      Miles, Seconds => 1.0);

   procedure Run (Object : Data_T) is
   begin
      Float_Io.Put (Object.Distance);
      Put (" " & Distance_T'Image (Object.Distance_Type) & " in ");
      Float_Io.Put (Object.Seconds);
      Put (" seconds = ");
      Float_Io.Put (C_Miles_Per_Hour (Object));
      Put_Line (" mph");
   end Run;

begin
   Run (Object_Feet);
   Run (Object_Meters);
   Run (Object_Miles);
end Main;
```

# Interfacing with C Lab Solution - C

```
enum DistanceT { FEET, METERS, MILES };
struct DataT {
    float distance;
    enum DistanceT distanceType;
    float seconds;
    };

float miles_per_hour (struct DataT data) {
   float miles = data.distance;
   switch (data.distanceType) {
      case METERS:
         miles = data.distance / 1609.344;
         break;
      case FEET:
         miles = data.distance / 5280.0;
         break;
   };
   return miles / (data.seconds / (60.0 * 60.0));
}
```

# Summary

# Summary

- Possible to interface with other languages (typically C)
- Ada provides some built-in support to make interfacing simpler
- Crossing languages can be made safer
    - But it still increases complexity of design / implementation

# Day 3 - PM

# 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

```
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

# 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

# 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

# Low Level Programming

# Introduction

# Introduction

- Sometimes you need to get your hands dirty
- Hardware Issues
  - Register or memory access
  - Assembler code for speed or size issues
- Interfacing with other software
  - Object sizes
  - Endianness
  - Data conversion

# Data Representation

# Data Representation Vs Requirements

- Developer usually defines requirements on a type

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

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

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

-- with representation clauses
type Another_Integer_T is range 1 .. 10;
for Another_Integer_T'Object_Size use 8;
for Another_Integer_T'Value_Size  use 4;
for Another_Integer_T'Alignment   use 1;
```

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

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

# Value_Size / Size

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

```
-- with aspect
type Small_T is range 1 .. 4
   with Size => 3;

-- with representation clause
type Another_Small_T is range 1 .. 4;
for Another_Small_T'Size use 3;
```

# Object Size (GNAT-Specific)

- `Object_Size` represents the size of the object in memory
- It must be a multiple of `Alignment * Storage_Unit (8)`, and at least equal to `Size`

```
-- with aspects
type Some_T is range 1 .. 4
   with Value_Size  => 3,
        Object_Size => 8;

-- with representation clauses
type Another_T is range 1 .. 4;
for Another_T'Value_Size use 3;
for Another_T'Object_Size use 8;
```

- Object size is the *default* size of an object, can be changed if specific representations are given

# Alignment

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

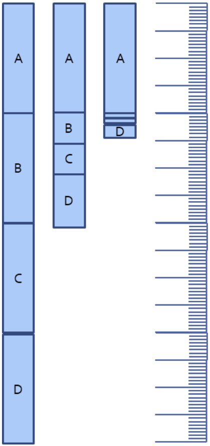
```
-- with aspects
type Aligned_T is range 1 .. 4
   with Size      => 4,
        Alignment => 8;

-- with representation clauses
type Another_Aligned_T is range 1 .. 4;
for Another_Aligned_T'Size use 4;
for Another_Aligned_T'Alignment use 8;
```

# Record Types

- Ada doesn't force any particular memory layout
- Depending on optimization of constraints, layout can be optimized for speed, size, or not optimized

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record;
```



# Pack Aspect

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

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record;
type Ar is array (1 .. 1000) of Boolean;
-- Rec'Size is 56, Ar'Size is 8000
```

- Packed

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record with Pack;
type Ar is array (1 .. 1000) of Boolean;
pragma Pack (Ar);
-- Rec'Size is 36, Ar'Size is 1000
```

# Enum Representation Clauses

- Can specify representation for each value
- Representation must have increasing number

```
type E is (A, B, C);
for E use (A => 2, B => 4, C => 8);
```

- Can use `E'Enum_Rep (A) = 2`
- Can use `E'Enum_Val (2) = A`

# Record Representation Clauses

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

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

# Unchecked Unions

- Allows replicating C's **union** with **discriminated** records
- Discriminant is **not stored**
- No discriminant check
- Object must be **mutable**

```
type R (Is_Float : Boolean := False) is record
    case Is_Float is
    when True =>
        F : Float;
    when False =>
        I : Integer;
    end case;
end record
    with Unchecked_Union;

O : R := (Is_Float => False, I => 1);
F : Float := R.F; --  no check!
```

# Array Representation Clauses

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

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

-- with representation clause
type Another_Array_T is array (1 .. 1000) of Boolean;
for Another_Array_T'Component_Size use 2;
```

# Endianness Specification

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

```
-- with aspect
type Array_T is array (1 .. 1000) of Boolean with
  Scalar_Storage_Order => System.Low_Order_First;

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

# Change of Representation

- Explicit new type can be used to set representation
- Very useful to unpack data from file/hardware to speed up references

```
type Rec_T is record
     Component1 : Unsigned_8;
     Component2 : Unsigned_16;
     Component3 : Unsigned_8;
end record;
type Packed_Rec_T is new Rec_T;
for Packed_Rec_T use record
   Component1 at 0 range  0 ..  7;
   Component2 at 0 range  8 .. 23;
   Component3 at 0 range 24 .. 31;
end record;
R : Rec_T;
P : Packed_Rec_T;
...
R := Rec_T (P);
P := Packed_Rec_T (R);
```

# Address Clauses and Overlays

# Address

- Ada distinguishes the notions of
  - A reference to an object
  - An abstract notion of address (`System.Address`)
  - The integer representation of an address
- Safety is preserved by letting the developer manipulate the right level of abstraction
- Conversion between pointers, integers and addresses are possible
- The address of an object can be specified through the `Address` aspect

# Address Clauses

- Ada allows specifying the address of an entity

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

Use_Rep_Clause : Unsigned_32;
for Use_Rep_Clause'Address use 16#5678_1234#;
```

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

```
Use_Aspect     : Unsigned_32 with
   Volatile,
   Address => 16#1234_ABCD#;

Use_Rep_And_Pragma : Unsigned_32;
for Use_Rep_And_Pragma'Address use 16#5678_1234#;
pragma Volatile (Use_Rep_And_Pragma);
```

- Useful to read a value anywhere

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

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

# Address Values

- The type `Address` is declared in `System`
  - But this is a `private` type
  - You cannot use a number
- Ada standard way to set constant addresses:
  - Use `System.Storage_Elements` which allows arithmetic on address

```
V : Unsigned_32 with
   Address =>
      System.Storage_Elements.To_Address (16#120#);
```

- GNAT specific attribute `'To_Address`
  - Handy but not portable

```
V : Unsigned_32 with
   Address => System'To_Address (16#120#);
```

# Volatile

- The `Volatile` property can be set using an aspect or a pragma
- Ada also allows volatile types as well as objects

```
type Volatile_U32 is mod 2**32 with Volatile;
type Volatile_U16 is mod 2**16;
pragma Volatile (Volatile_U16);
```

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

# Ada Address Example

```
type Bit_Array_T is array (Integer range <>) of Boolean
   with Component_Size => 1;

-- objects can be referenced elsewhere
Object  : aliased Integer with Volatile;
Object2 : aliased Integer with Volatile;

Object_A : System.Address := Object'Address;
Object_I : Integer_Address := To_Integer (Object_A);

--  This overlays Bit_Array_Object onto Object in memory
Bit_Array_Object : aliased Bit_Array_T (1 .. Object'Size)
   with Address => Object_A;

Object2_Alias : aliased Integer
   --  Trust me, I know what I'm doing, this is Object2
   with Address => To_Address (Object_I - 4);
```

# Aliasing Detection

- *Aliasing*: multiple objects are accessing the same address
  - Types can be different
  - Two pointers pointing to the same address
  - Two references onto the same address
  - Two objects at the same address
- `Var1'Has_Same_Storage (Var2)` checks if two objects occupy exactly the same space
- `Var'Overlaps_Storage (Var2)` checks if two object are partially or fully overlapping

# Unchecked Conversion

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

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

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

# Tricks

# Package Interfaces

- Package `Interfaces` provide Integer and unsigned types for many sizes
  - `Integer_8`, `Integer_16`, `Integer_32`, `Integer_64`
  - `Unsigned_8`, `Unsigned_16`, `Unsigned_32`, `Unsigned_64`
- With shift/rotation functions for unsigned types

# Fat/Thin Pointers for Arrays

- Unconstrained array access is a fat pointer

```
type String_Acc is access String;
Msg : String_Acc;
-- array bounds stored outside array pointer
```

- Use a size representation clause for a thin pointer

```
type String_Acc is access String;
for String_Acc'Size use 32;
-- array bounds stored as part of array pointer
```

# Flat Arrays

- A constrained array access is a thin pointer
  - No need to store bounds

```
type Line_Acc is access String (1 .. 80);
```

- You can use big flat array to index memory
  - See `GNAT.Table`
  - Not portable

```
type Char_array is array (natural) of Character;
type C_String_Acc is access Char_Array;
```

# Lab

# Low Level Programming Lab

(Simplified) Message generation / propagation

- Overview
  - Populate a message structure with data and a CRC (cyclic redundancy check)
  - "Send" and "Receive" messages and verify data is valid
- Goal
  - You should be able to create, "send", "receive", and print messages
  - Creation should include generation of a CRC to ensure data security
  - Receiving should include validation of CRC

# Project Requirements

- Message Generation
  - Message should at least contain:
    - Unique Identifier
    - (Constrained) string component
    - Two other components
    - CRC value
- "Send" / "Receive"
  - To simulate send/receive:
    - "Send" should do a byte-by-byte write to a text file
    - "Receive" should do a byte-by-byte read from that same text file
  - Receiver should validate received CRC is valid
    - You can edit the text file to corrupt data

# Hints

- Use a representation clause to specify size of record
    - To get a valid size, individual components may need new types with their own rep spec
- CRC generation and file read/write should be similar processes
    - Need to convert a message into an array of "something"

# Low Level Programming Lab Solution - CRC

```
with System;
package Crc is
   type Crc_T is mod 2**32;
   for Crc_T'Size use 32;
   function Generate
     (Address : System.Address;
      Size : Natural)
      return Crc_T;
end Crc;

package body Crc is
   type Array_T is array (Positive range <>) of Crc_T;
   function Generate
     (Address : System.Address;
      Size    : Natural)
      return Crc_T is
      Word_Count : Natural;
      Retval     : Crc_T := 0;
   begin
      if Size > 0
      then
         Word_Count := Size / 32;
         if Word_Count * 32 /= Size
         then
            Word_Count := Word_Count + 1;
         end if;
         declare
            Overlay : Array_T (1 .. Word_Count);
            for Overlay'Address use Address;
         begin
            for I in Overlay'Range
            loop
               Retval := Retval + Overlay (I);
            end loop;
         end;
      end if;
      return Retval;
   end Generate;
end Crc;
```

# Low Level Programming Lab Solution - Messages (Spec)

```
with Crc; use Crc;
package Messages is
   type Message_T is private;
   type Command_T is (Noop, Direction, Ascend, Descend, Speed);
   for Command_T use
     (Noop => 0, Direction => 1, Ascend => 2, Descend => 4, Speed => 8);
   for Command_T'Size use 8;
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T;
   function Get_Crc (Message : Message_T) return Crc_T;
   procedure Write (Message : Message_T);
   procedure Read (Message : out Message_T;
                   valid : out boolean);
   procedure Print (Message : Message_T);
private
   type U32_T is mod 2**32;
   for U32_T'Size use 32;
   Max_Text_Length : constant := 20;
   type Text_Index_T is new Integer range 0 .. Max_Text_Length;
   for Text_Index_T'Size use 8;
   type Text_T is record
      Text : String (1 .. Max_Text_Length);
      Last : Text_Index_T;
   end record;
   for Text_T'Size use Max_Text_Length * 8 + Text_Index_T'size;
   type Message_T is record
      Unique_Id : U32_T;
      Command   : Command_T;
      Value     : U32_T;
      Text      : Text_T;
      Crc       : Crc_T;
   end record;
end Messages;
```

# Low Level Programming Lab Solution - Main (Helpers)

```
with Ada.Text_IO; use Ada.Text_IO;
with Messages;
procedure Main is
   Message : Messages.Message_T;
   function Command return Messages.Command_T is
   begin
      loop
         Put ("Command (");
         for E in Messages.Command_T
         loop
            Put (Messages.Command_T'Image (E) & " ");
         end loop;
         Put ("): ");
         begin
            return Messages.Command_T'Value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Command;
   function Value return Positive is
   begin
      loop
         Put ("Value: ");
         begin
            return Positive'Value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Value;
   function Text return String is
   begin
      Put ("Text: ");
      return Get_Line;
   end Text;
```

# Low Level Programming Lab Solution - Main

```
   procedure Create is
      C : constant Messages.Command_T := Command;
      V : constant Positive           := Value;
      T : constant String             := Text;
   begin
      Message := Messages.Create
          (Command => C,
           Value   => V,
           Text    => T);
   end Create;
   procedure Read is
      Valid : Boolean;
   begin
      Messages.Read (Message, Valid);
      Ada.Text_IO.Put_Line("Message valid: " & Boolean'Image (Valid));
   end read;
begin
   loop
      Put ("Create Write Read Print: ");
      declare
         Command : constant String := Get_Line;
      begin
         exit when Command'Length = 0;
         case Command (Command'First) is
            when 'c' | 'C' =>
               Create;
            when 'w' | 'W' =>
               Messages.Write (Message);
            when 'r' | 'R' =>
               read;
            when 'p' | 'P' =>
               Messages.Print (Message);
            when others =>
               null;
         end case;
      end;
   end loop;
end Main;
```

# Low Level Programming Lab Solution - Messages (Helpers)

```
with Ada.Text_IO;
with Unchecked_Conversion;
package body Messages is
   Global_Unique_Id : U32_T := 0;
   function To_Text (Str : String) return Text_T is
      Length : Integer := Str'Length;
      Retval : Text_T  := (Text => (others => ' '), Last => 0);
   begin
      if Str'Length > Retval.Text'length then
         Length := Retval.Text'Length;
      end if;
      Retval.Text (1 .. Length) := Str (Str'First .. Str'first + Length - 1);
      Retval.Last               := Text_Index_T (Length);
      return Retval;
   end To_Text;
   function From_Text (Text : Text_T) return String is
      Last : constant Integer := Integer (Text.Last);
   begin
      return Text.Text (1 .. Last);
   end From_Text;
   function Get_Crc (Message : Message_T) return Crc_T is
   begin
      return Message.Crc;
   end Get_Crc;
   function Validate (Original : Message_T) return Boolean is
      Clean : Message_T := Original;
   begin
      Clean.Crc := 0;
      return Crc.Generate (Clean'Address, Clean'Size) = Original.Crc;
   end Validate;
```

# Low Level Programming Lab Solution - Messages (Body)

```ada
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T is
      Retval : Message_T;
   begin
      Global_Unique_Id := Global_Unique_Id + 1;
      Retval           :=
        (Unique_Id => Global_Unique_Id, Command => Command,
         Value     => U32_T (Value), Text => To_Text (Text), Crc => 0);
      Retval.Crc := Crc.Generate (Retval'Address, Retval'Size);
      return Retval;
   end Create;
   type Char is new Character;
   for Char'Size use 8;
   type Overlay_T is array (1 .. Message_T'Size / 8) of Char;
   function Convert is new Unchecked_Conversion (Message_T, Overlay_T);
   function Convert is new Unchecked_Conversion (Overlay_T, Message_T);
   Const_Filename : constant String := "message.txt";
   procedure Write (Message : Message_T) is
      Overlay : constant Overlay_T := Convert (Message);
      File    : Ada.Text_IO.File_Type;
   begin
      Ada.Text_IO.Create (File, Ada.Text_IO.Out_File, Const_Filename);
      for I in Overlay'Range loop
         Ada.Text_IO.Put (File, Character (Overlay (I)));
      end loop;
      Ada.Text_IO.New_Line (File);
      Ada.Text_IO.Close (File);
   end Write;
   procedure Read (Message : out Message_T;
                   Valid   : out Boolean) is
                   Overlay : Overlay_T;
                   File    : Ada.Text_IO.File_Type;
   begin
      Valid := False;
      Ada.Text_IO.Open (File, Ada.Text_IO.In_File, Const_Filename);
      declare
         Str : constant String := Ada.Text_IO.Get_Line (File);
      begin
         Ada.Text_IO.Close (File);
         for I in Str'Range loop
            Overlay (I) := Char (Str (I));
         end loop;
         Message := Convert (Overlay);
         Valid   := Validate (Message);
      end;
   end Read;
   procedure Print (Message : Message_T) is
   begin
      Ada.Text_IO.Put_Line ("Message" & U32_T'Image (Message.Unique_Id));
      Ada.Text_IO.Put_Line ("  " & Command_T'Image (Message.Command) & " =>" &
                            U32_T'Image (Message.Value));
      Ada.Text_IO.Put_Line ("  Additional Info: " & From_Text (Message.Text));
   end Print;
end Messages;
```

# Summary

# Summary

- Like C, Ada allows access to assembly-level programming
- Unlike C, Ada imposes some more restrictions to maintain some level of safety
- Ada also supplies language constructs and libraries to make low level programming easier

# Supplementary Resource: Inline ASM

# Calling Assembly Code

- Calling assembly code is a vendor-specific extension
- GNAT allows passing assembly with `System.Machine_Code.ASM`
  - Handled by the linker directly
- The developer is responsible for mapping variables on temporaries or registers
- See documentation
  - GNAT RM 13.1 Machine Code Insertion
  - GCC UG 6.39 Assembler Instructions with C Expression Operands

# Simple Statement

- Instruction without inputs/outputs

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

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

- You can group several instructions

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

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

# Operands

- It is often useful to have inputs or outputs...
  - `Asm_Input` and `Asm_Output` attributes on types

First operand (starting with o)

Use %% for %

Input operand for type Unsigned_8

Constraint, 'a' for eax

Value

```
Asm ("outb %0,%%dx",
     Inputs => (Unsigned_8'Asm_Input ("a", B),
                Port_Id'Asm_Input ("d", P)),
     Volatile => True);
```

Instruction

Constraint, 'd' for edx

Name

# Mapping Inputs / Outputs on Temporaries

```
Asm (<script referencing $<input> >,
     Inputs  => ({<type>'Asm_Input (<constraint>,
                                     <variable>)}),
     Outputs => ({<type>'Asm_Output (<constraint>,
                                      <variable>)});
```

- **assembly script** containing assembly instructions + references to registers and temporaries
- **constraint** specifies how variable can be mapped on memory (see documentation for full details)

| Constraint | Meaning |
|---|---|
| R | General purpose register |
| M | Memory |
| F | Floating-point register |
| I | A constant |
| g | global (on x86) |
| a | eax (on x86) |

# Main Rules

- No control flow between assembler statements
  - Use Ada control flow statement
  - Or use control flow within one statement
- Avoid using fixed registers
  - Makes compiler's life more difficult
  - Let the compiler choose registers
  - You should correctly describe register constraints
- On x86, the assembler uses `AT&T` convention
  - First operand is source, second is destination
- See your toolchain's `as` assembler manual for syntax

# Volatile and Clobber ASM Parameters

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

# Instruction Counter Example (x86)

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

# Reading a Machine Register (ppc)

```
function Get_MSR return MSR_Type is
   Res : MSR_Type;
begin
   Asm ("mfmsr %0",
        Outputs => MSR_Type'Asm_Output ("=r", Res),
        Volatile => True);
   return Res;
end Get_MSR;
generic
    Spr : Natural;
 function Get_Spr return Unsigned_32;
 function Get_Spr return Unsigned_32 is
    Res : Unsigned_32;
 begin
    Asm ("mfspr %0,%1",
         Inputs => Natural'Asm_Input ("K", Spr),
         Outputs => Unsigned_32'Asm_Output ("=r", Res),
         Volatile => True);
    return Res;
 end Get_Spr;
 function Get_Pir is new Get_Spr (286);
```

# Writing a Machine Register (ppc)

```
generic
   Spr : Natural;
procedure Set_Spr (V : Unsigned_32);
procedure Set_Spr (V : Unsigned_32) is
begin
   Asm ("mtspr %0,%1",
        Inputs => (Natural'Asm_Input ("K", Spr),
                   Unsigned_32'Asm_Input ("r", V)));
end Set_Spr;
```

# Day 4 - AM

# GNAT Project Facility Overview

# Introduction

# Origins and Purposes of Projects

- Need for flexibility
    - Managing huge applications is a difficult task
    - Build tools are always useful
- GNAT compilation model
    - Compiler needs to know where to find Ada files imported by Ada unit being compiled
- IDEs
    - AdaCore IDEs need to know where to find source and object files
- Tools (metrics, documentation generator, etc)
    - AdaCore tools benefit from having knowledge of application structure

# Subsystems of Subsystems of ...

- Projects support incremental/modular project definition
  - Projects can import other projects containing needed files
  - Child projects can extend parent projects
    - Inheriting all attributes of parent
    - Allows override of source files and other attributes
- Allows structuring of large development efforts into hierarchical subsystems
  - Build decisions deferred to subsystem level

# Project Files

# GNAT Project Files

- Text files with Ada-like syntax
- Also known as *GPR files* due to file extension
- Integrated into command-line tools
  - Specified via the `-P project-file-name` switch
- Integrated into IDEs
  - A fundamental part
  - Automatically generated if desired
- Should be under configuration management

# Configurable Properties

- Source directories and specific files' names
- Output directory for object modules and .ali files
- Target directory for executable programs
- Switch settings for project-enabled tools
- Source files for main subprogram(s) to be built
- Source programming languages
    - Ada / C / C++ are preconfigured
- Source file naming conventions
- And many more

# The Minimal Project File

```
project My_Project is
end My_Project;
```

# Specifying Main Subprogram(s)

- Optional
  - Some projects do not build an executable
  - If necessary and not specified in file, must be specified on command-line
- Can have more than one file named
- A project-level setting

```
project Foo is
   for Main use ("bar.adb", "baz.adb");
end Foo;
```

# About Project Files and Makefiles

- A Makefile performs actions (indirectly)
- A project file describes a project
- Command lines using project files fit naturally in Makefile paradigm

```
gprbuild -P <project-file> ...
```

# Building with GPRbuild

# Introduction

# Generic Build Tool

- Designed for construction of large multi-language systems
  - Allows subsystems and libraries
- Manages three step build process:
  - Compilation phase:
    - Each compilation unit examined in turn, checked for consistency, and, if necessary, compiled (or recompiled) as appropriate
  - Post-compilation phase (binding):
    - Compiled units from a given language are passed to language-specific post-compilation tool (if any)
    - Objects grouped into static or dynamic libraries as specified
  - Linking phase:
    - Units or libraries from all subsystems are passed to appropriate linker tool

# Command Line

# GPRbuild Command Line

- Made up of three components
  - Main project file (required)
  - Switches (optional)
    - `gprbuild` switches
    - Options for called tools
  - Main source files (optional)
    - If not specified, executable(s) specified in project file are built
    - If no main files specified, no executable is built

# Common Options Passed To Tools

`-cargs options`

- Options passed to all compilers
- Example:

```
-cargs -g
```

`-cargs:<language> options`

- Options passed to compiler for specific language
- Examples:

```
-cargs:Ada -gnatf
-cargs:C -E
```

`-bargs options`

- Options passed to all binder drivers

`-bargs:<language> options`

- Options passed to binder driver for specific language
- Examples:

```
-bargs:Ada binder_prefix=ppc-elf
-bargs:C++ c_compiler_name=ccppc
```

`-largs options`

- Options passed to linker for generating executable

# Common Command Line Switches

| | |
|---|---|
| `-P <project file>` | Name of main project file (space between *P* and <*filename*> is optional) |
| `-aP <directory>` | Add <*directory*> to list of directories to search for project files |
| `-u [<source file> [, <source file>...]]` | If sources specified, only compile these sources.<br>Otherwise, compile all sources in main project file |
| `-U [<source file> [, <source file>...]]` | If sources specified, only compile these sources.<br>Otherwise, compile all sources in project tree |
| `-Xnm=val` | Specify external reference that may be read via built-in function `external`. |
| `--version` | Display information about GPRbuild: version, origin and legal status |
| `--help` | Display GPRbuild usage |
| `--config=<config project file name>` | Configuration project file name (default `default.cgpr`) |

# Common Build Switches

Switches to be specified on command line or in `Builder` package of main project file

| | |
|---|---|
| `--create-map-file[=<map file>]` | When linking, (if supported) by the platform, create a map file `<map file>`. (If not specified, filename is `<executable name>.map`) |
| `-j<num>` | Use <num> simultaneous compilation jobs |
| `-k` | Keep going after compilation errors (default is to stop on first error) |
| `-p (or --create-missing-dirs)` | Creating missing output directory (e.g. object directory) |

# Lab

# Start GPRbuild

- Open a command shell
- Go to `gpr_1_building_with_gprbuild` directory (under `source`)
  - Contains a main procedure and a supporting package for the "8 Queens" problem
- Use an editor to create minimum project file
  - Name the project anything you wish
  - Filename and project name should be the same
- Build `Queens` using `gprbuild` and the project file as-is
  - Use `-P` argument on the command line to specify project file
  - Must also specify file name on command line to get executable
    - For example: `gprbuild -P lab.gpr queens`
- Clean the project with `gprclean`
  - Use `-P` argument on the command line to specify project file
  - Note that the `queens.exe` executable remains
    - Plus (possibly) some intermediate files

# GPRbuild Lab - Simple GPR File

```
project Lab is
end Lab;
```

`gprbuild -P lab.gpr` Only compiles source files

`gprbuild -P lab.gpr queens` Compiles source and creates `queens` executable

`gprclean -P lab.gpr` Deletes ALI and object files for `Queens` and `Queens_Pkg`

# GPRbuild Lab Part 2

- Change project file so that it specifies the main program
- Build again, without specifying the main on the command line
  - Use only `-P` argument on the command line to specify project file
- Clean the project with `gprclean` again
  - Note the `queens` executable is now also deleted (as well as any intermediate files)

# GPRbuild Lab - Main Program Specified

```
project Lab is
   for Main use ( "main.adb" );
end Lab;
```

`gprbuild -P lab.gpr` Compiles source and creates `queens` executable

`gprclean -P lab.gpr` Deletes all generated files

# Project Properties

# Introduction

# Specifying Directories

- Any number of Source Directories
  - Source Directories contain source files
  - If not specified, defaults to directory containing project file
  - Possible to create a project with no Source Directory
    - Not the same as not specifying the Source Directory!
- One Object Directory
  - Contains object files and other tool-generated files
  - If not specified, defaults to directory containing project file
- One Executables Directory
  - Contains executable(s)
  - If not specified, defaults to same location as Object Directory
- *Tip: use forward slashes rather than backslashes for the most portability*
  - Backslash will only work on Windows
  - Forward slash will work on all supported systems (including Windows)

# Variables

**Typed** Set of possible string values

**Untyped** Unspecified set of values (strings and lists)

```
project Build is
   type Targets is ("release", "test");
   -- Typed variable
   Target : Targets := external("target", "test");
   -- Untyped string variable
   Var := "foo";
   -- Untyped string list variable
   Var2 := ("-gnato", "-gnata");
   ...
end Build;
```

# Typed Versus Untyped Variables

- Typed variables have only listed values possible
  - Case sensitive, unlike Ada
- Typed variables are declared once per scope
  - Once at project or package level
  - Essentially read-only constants
    - Useful for external inputs
- Untyped variables may be "declared" many times
  - No previous declaration required

# Property Values

- Strings
- Lists of strings

  ```
  ("-v", "-gnatv")
  ```

- Associative arrays
  - Map input string to either single string or list of strings

    *for <name> (<string-index>) use <list-of_strings>;*

    ```
    for Switches ("Ada") use ("-gnaty", "-gnatwa");
    ```

# Directories

# Source Directories

- One or more in any project file
- Default is same directory as project file
- Can specify additional / other directories

```
for Source_Dirs use ("src/mains", "src/drivers", "foo");
```

- Can specify an entire tree of directories

```
for Source_Dirs use ("src/**");
```

  - `src` directory and every subdirectory underneath

# Source Files

- Must be at least one **immediate** source file
  - *Immediate*
    - Resides in project source directories OR
    - Specified through source-related attribute
  - Unless explicitly specified none present

    ```
    for Source_Files use ();
    ```
- Can specify source files by name

  ```
  for Source_Files use ("pack1.ads","pack2.adb");
  ```
- Can specify an external file containing source names

  ```
  for Source_List_File use "source_list.txt";
  ```

# Object Directory

- Specifies location for files generated by compiler (or tools)
  - Such as `.ali` files and `.o` files
  - For the project's immediate sources

```
project Release is
   for Object_Dir use "release";
   ...
end Release;
```

- Only one object directory per project

# Executable Directory

- Specifies the location for executable image

```
project Release is
   for Exec_Dir use "executables";
   ...
end Release;
```

- Default is same directory as object files
- Only one per project

# Project Packages

# Packages Correspond to Tools

- Packages within project file contain switches (generally) for specific tools
- Allowable names and content defined by vendor
  - Not by users

| | | |
|---|---|---|
| Analyzer | Binder | Builder |
| Check | Clean | Compiler |
| Cross_Reference | Documentation | Eliminate |
| Finder | Gnatls | Gnatstub |
| IDE | Install | Linker |
| Metrics | Naming | Pretty_Printer |
| Remote | Stack | Synchronize |

# Setting Tool Switches

- May be specified to apply by default

```
package Compiler is
   for Default_Switches ("Ada") use ("-gnaty", "-v");
end Compiler;
```

- May be specified on per-unit basis
  - Associative array "Switches" indexed by unit name

```
package Builder is
   for Switches ("main1.adb") use ("-O2");
   for Switches ("main2.adb") use ("-g");
end Builder;
```

# Naming Considerations

# Rationale

- Project files assume source files have GNAT naming conventions

  Specification `<unitname>[-<childunit>].ads`

  Body `<unitname>[-<childunit>].adb`
- Sometimes you want different conventions
  - Third-party libraries
  - Legacy code used different compiler
  - Changing filenames would make tracking changes harder

# Source File Naming Schemes

- Allow arbitrary naming conventions
  - Other than GNAT default convention
- Specified in a package named `Naming`
  - May be applied to all source files in a project
  - May be applied to specific files in a project
    - Individual attribute specifications

# Foreign Default File Naming Example

- Sample source file names
  - Package spec for `Utilities` in **`utilities.spec`**
  - Package body for `Utilities` in **`utilities.body`**
  - Package spec for `Utilities.Child` in **`utilities.child.spec`**
  - Package body for `Utilities.Child` in **`utilities.child.body`**

```
project Legacy_Code is
   ...
   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use ".";
      for Spec_Suffix ("Ada") use ".spec";
      for Body_Suffix ("Ada") use ".body";
   end Naming;
   ...
end Legacy_Code;
```

# GNAT Default File Naming Example

- Sample source file names
  - Package spec for `Utilities` in `utilities.ads`
  - Package body for `Utilities` in `utilities.adb`
  - Package spec for `Utilities.Child` in `utilities-child.ads`
  - Package body for `Utilities.Child` in `utilities-child.adb`

```
project GNAT is
   ...
   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use "-";
      for Spec_Suffix ("Ada") use ".ads";
      for Body_Suffix ("Ada") use ".adb";
   end Naming;
   ...
end GNAT;
```

# Individual (Arbitrary) File Naming

- Uses associative arrays to specify file names
  - Index is a string containing the unit name
    - Case insensitive
  - Value is a string containing the file name
    - Case sensitivity depends on host file system
- Has distinct attributes for specs and bodies
  - Syntax: *for Spec ("<Ada unit name>") use "<filename>";*

```
for Spec ("MyPack.MyChild") use "MMS1AF32.A";
for Body ("MyPack.MyChild") use "MMS1AF32.B";
```

# Variables for Conditional Processing

# Two Sample Projects for Different Switch Settings

```
project Debug is
  for Object_Dir use "debug";
  package Builder is
    for Default_Switches ("Ada")
       use ("-g");
  end Builder;
  package Compiler is
    for Default_Switches ("Ada")
       use ("-fstack-check",
            "-gnata",
            "-gnato");
  end Compiler;
end Debug;
```

```
project Release is
  for Object_Dir use "release";
  package Compiler is
    for Default_Switches ("Ada")
       use ("-O2");
  end Compiler;
end Release;
```

# External and Conditional References

- Allow project file content to depend on value of environment variables and command-line arguments
- Reference to external values is by function

  `external (<name> [, default])`

  - Returns value of **name** as supplied via
    - Command line
    - Environment variable
    - If not specified, uses **default** or else ""
- Command line switch
  - Syntax: *gprbuild -P... -Xname=value ...*

```
gprbuild -P common/build.gpr -Xtarget=test common/main.adb
```

**Note**

Command line values override environment variables

# External/Conditional Reference Example

```
project Build is
   type Targets is ("release", "test");
   Target : Targets := external("target", "test");
   case Target is -- project attributes
      when "release" =>
         for Object_Dir use "release";
         for Exec_Dir use ".";
      when "test" =>
         for Object_Dir use "debug";
   end case;
   package Compiler is
      case Target is
         when "release" =>
            for Default_Switches ("Ada") use ("-O2");
         when "test" =>
            for Default_Switches ("Ada") use
                  ("-g", "-fstack-check", "-gnata", "-gnato");
      end case;
   end Compiler;
   ...
end Build;
```

# Scenario Controlling Source File Selection

```
project Demo is
   ...
   type Displays is ("Win32", "ANSI");
   Output : Displays := external ("OUTPUT", "Win32");
   ...
   package Naming is
      case Output is
         when "Win32" =>
            for Body ("Console") use "console_win32.adb";
         when "ANSI" =>
            for Body ("Console") use "console_ansi.adb";
      end case;
   end Naming;
end Demo;
```

- Source Files

| console.ads | console_win32.adb | console_ansi.adb |
|---|---|---|
| `package Console is`<br>`...`<br>`end Console;` | `package body Console is`<br>`...`<br>`end Console;` | `package body Console is`<br>`...`<br>`end Console;` |

# Lab

# Project Properties Lab

- Create new project file in an empty directory
- Specify source and output directories
  - Use source files from the **`gpr_2_project_properties`** directory (under **`source`**)
  - Specify where object files and executable should be located
- Build and run executable (pass command line argument of 200)
  - Note location of object files and executable
  - Execution should get `Constraint_Error`

# Directories Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties");
   for Main use ( "main.adb" );
   for Object_Dir use "obj";
   for Exec_Dir use "exec";
end Lab;
```

- Executable Output

```
...
41     267914296
42     433494437
43     701408733
44    1134903170
45    1836311903

raised CONSTRAINT_ERROR : fibonacci.adb:16 overflow check failed
```

# Project Properties Lab (1/3) - Switches

- Modify project file to disable overflow checking
  - Add the `Compiler` package
  - Insert `Default_Switches` attribute for Ada in `Compiler` package
  - Set switch `-gnato0` in the attribute
    - Disable overflow checking
- Build and run again
  - Need to use switch `-f` on command line to force rebuild
    - (Changes to GPR file do not automatically force recompile)
  - No `Constraint_Error`
    - But data doesn't look right due to overflow issues

# Switches Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties");
   for Main use ( "main.adb" );

   package Compiler is
      for Default_Switches ("Ada") use ("-gnato0");
   end Compiler;
   ...
end Lab;
```

- Executable Output

```
...
43    701408733
44   1134903170
45   1836311903
46  -1323752223
47    512559680
48   -811192543
49   -298632863
50  -1109825406
...
```

# Project Properties Lab (2/3) - Naming

- Modify project file to use naming conventions from a different compiler
  - Change source directories to point to `naming` folder
  - File naming conventions:
    - Spec: `<unitname>[.child].1.ada`
    - Body: `<unitname>[.child].2.ada`
  - Remember to fix executable name
- Build and run again
  - *Note: Accumulator uses more bits, so failure condition happens later*

# Naming Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties/naming");

   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use ".";
      for Spec_Suffix ("Ada") use ".1.ada";
      for Body_Suffix ("Ada") use ".2.ada";
   end Naming;

   for Main use ( "main.2.ada" );
   ...
end Lab;
```

- Executable Output

```
...
88   1779979416004714189
89   2880067194370816120
90   4660046610375530309
91   7540113804746346429
92  -6246583658587674878
93   1293530146158671551
94  -4953053512429003327
95  -3659523366270331776
96  -8612576878699335103
...
```

# Project Properties Lab (3/3) - Conditional

- Modify project file to select precision via compiler switch
  - `conditional` folder has two more package bodies using different accumulators
  - Read a variable from the command line to determine which body to use
    - Hint: `Naming` will need to use a `case` statement to select appropriate body
- Build and run again
  - Hint: Name used in **external** call must be same casing as in GPRBUILD command, i.e
    - `external ("FooBar");` means `gprbuild -XFooBar...`

# Conditional Solution

- Project File

```
project Lab is

   for Source_Dirs use ("source/030_project_properties/naming",
                        "source/030_project_properties/conditional");

   type Precision_T is ( "unsigned", "float", "default" );
   Precision : Precision_T := external ( "PRECISION", "default");

   package Naming is
   ...
      case Precision is
      when "unsigned" =>
         for Body ("Fibonacci") use "fibonacci.unsigned";
      when "float" =>
         for Body ("Fibonacci") use "fibonacci.float";
      when "default" =>
         for Body ("Fibonacci") use "fibonacci.2.ada";
      end case;
   end Naming;

   ...
end Lab;
```

- Executable Output

```
1    1.00000000000000E+00
2    2.00000000000000E+00
3    3.00000000000000E+00
4    5.00000000000000E+00
5    8.00000000000000E+00
6    1.30000000000000E+01
7    2.10000000000000E+01
8    3.40000000000000E+01
9    5.50000000000000E+01
10    8.90000000000000E+01
...
```

# Structuring Your Application

# Introduction

# Introduction

- Most applications can be broken into pieces
  - Modules, components, etc - whatever you want to call them
- Helpful to have a project file for each component
  - Or even multiple project files for better organization

# Dependency

- Units of one component typically depend units in other components
  - Types packages, utilities, external interfaces, etc
- A project can `with` another project to allow visibility
  - Ambiguity issues can occur if the same unit appears in multiple projects

# Extension

- Sometimes we want to replace units for certain builds
  - Testing might require different package bodies
  - Different targets might require different values for constants
- A project can *extend* another project
  - Project inherits properties and units from its parent
  - Project can create new properties and units to override parent

# Building an Application

# Importing Projects

- Source files of one project may depend on source files of other projects
    - *Depend* in Ada sense (contains `with` clauses)
- Want to localize properties of other projects
    - Switches etc.
    - Defined in one place and not repeated elsewhere
- Thus dependent projects *import* other projects to add source files to search path

# Project Import Notation

- Similar to Ada's **with** clauses
  - But uses strings

```
with <literal string> {, <literal  string>};
```

- String literals are path names of project files
  - Relative
  - Absolute

```
with "/gui/gui.gpr", "../math.gpr";
project MyApp is
   ...
end MyApp;
```

# GPRbuild search paths

GPR with **relative** paths are searched

- From the current project directory
- From the environment variables
  - Path to a file listing directory paths
    - `GPR_PROJECT_PATH_FILE`
  - List of directories, separated by PATH-like (`:`, `;`) separator
    - `GPR_PROJECT_PATH`
- From the current toolchain's installation directory
  - Can be target-specific
  - Can be runtime-specific
  - See GPR Tool's User Guide

# Importing Projects Example

```
with GUI, Math;
package body Pack is
   ...
```

- Source Architecture

| /gui | | /myapp | | /math |
|---|---|---|---|---|
| gui.gpr | → | myapp.gpr | ← | math.gpr |
| gui.ads | | pack.ads | | math.ads |
| gui.adb | | pack.adb | | math.adb |
| | | main.adb | | |

- Project File

```
with "/gui/gui.gpr", "/math/math.gpr";
project MyApp is
   ...
end MyApp;
```

# Referencing Imported Content

- When referencing imported projects, use the Ada *dot notation* concept for declarations
  - Start with the project name
  - Use the tick (') for attributes

```
with "foo.gpr";
project P is
   package Compiler is
      for Default_Switches ("Ada") use
         Foo.Compiler'Default_Switches("Ada") & "-gnatwa";
   end Compiler;
end P;
```

  - Project P uses all the compiler switches in project Foo and adds `-gnatwa`

**Note**

in GPR files, "&" can be used to concatenate string lists and strings

# Renaming

- Packages can rename imported packages
- Effect is as if the package is declared locally
  - Much like the Ada language

```
with "../naming_schemes/rational.gpr";
project Clients is
   package Naming renames Rational.Naming;
   for Languages use ("Ada");
   for Object_Dir use ".";
   ...
end Clients;
```

# Project Source Code Dependencies

- Not unusual for projects to be interdependent
  - In the `Nav` project

```
with Hmi.Controls;
package body Nav.Engine is
   Global_Speed : Speed_T := 0.0;
   procedure Increase
      (Change : Speed_Delta_T) is
   begin
      Global_Speed := Global_Speed + Change;
      Hmi.Controls.Display (Global_Speed);
   end Increase;
end Nav.Engine;
```

  - In the `HMI` project

```
package body Hmi.Controls is
   procedure Display
      (Speed : Nav.Engine.Speed_T) is
   begin
      Display_On_Console (Speed);
   end Display;
   procedure Change
      (Speed_Change : Nav.Engine.Speed_Delta_T) is
   begin
      Nav.Engine.Increase (Speed_Change);
   end Change;
end Hmi.Controls;
```

# Project Dependencies

- Project files cannot create a cycle using **with**
  - Neither direct (Hmi → Nav → Hmi)
  - Nor indirect (Hmi → Nav → Monitor → Hmi)
- So how do we allow the sources in each project to interact?
  - **limited with**
  - Allows sources to be interdependent, but not the projects

```
limited with "Hmi.gpr";
project Nav is
  package Compiler is
    for Switches ("Ada") use
        Hmi.Compiler'Switches & "-gnatwa"; -- illegal
  end Compiler;
end Nav;
```

# Subsystems

- Sets of sources and folders managed together
- Represented by project files
  - Connected by project *with clauses* or project extensions
  - Generally one **primary** project file
  - Potentially many project files, assuming subsystems composed of other subsystems
- Have at most one *objects* folder per subsystem
  - A defining characteristic
  - Typical, not required

# Subsystems Example

```
with "gui.gpr";
with "utilities.gpr";
with "hardware.gpr";
project Application is
   for Main use ("demo");
   for Object_Dir use ("objs");
   ...
end Application;

with "utilities.gpr";
project Gui is
   for Object_Dir use ("objs");
   ...
end Gui;

with "utilities.gpr";
project Hardware is
   for Object_Dir use ("objs");
   ...
end Hardware;

project Utilities is
   for Object_Dir use ("objs");
   ...
end Utilities;
```

# Building Subsystems

- One project file given to the builder
- Everything necessary will be built, transitively
  - Build `Utilities`
    - Only source specified in `utilities.gpr` will be built
  - Build `Hardware` (or `Gui`)
    - Source specified in `hardware.gpr` (or `gui.gpr`) will be built
    - Source specified in `utilities.gpr` will be built if needed
  - Build `Application`
    - Any source specified in any of the project files will be built as needed

# Extending Projects

# Extending Projects

- Allows using modified versions of source files without changing the original sources
- Based on *inheritance* of parent project's properties
  - Source files
  - Switch settings
- Supports localized build decisions and properties
  - Inherited properties may be overridden with new versions
- Hierarchies permitted

# Project Extension

```
project Child extends "parent.gpr" is
```

- New project `Child` inherits everything from `Parent`
  - Except whatever new source/properties are specified in `Child`
- When compiling project `Child`
  - Source files in `Child` get compiled into its object directory
  - For source files in `Parent` that are not overridden in `Child`
    - If the source file is compiled into the `Parent` object directory, that file is considered compiled
    - If the source file is not compiled into the `Parent` object directory, that file will be compiled into the `Child` object directory

# Limits on Extending Projects

- A project that extends/modifies a project can also import other projects.
- Can't import both a parent and a modified project.
  - If you import the extension, you get the parent
- Can extend only one other project at a time.

# Multiple Versions of Unit Bodies Example

- Assume *Baseline* directory structure:
  - `baseline.gpr` contains
    - `filename.ads`
    - `filename.adb`
    - `application.adb`
- For testing, you want to
  - Replace `filename.adb` with a dummy version
  - Use `test_driver.adb` as the main program

# Multiple Versions of Unit Bodies Files

- *Baseline* GPR file might look like:

```
project Baseline is
   for Source_Dirs use ("src");
   for Main use ("application");
end Baseline;
```

- Test GPR file might look like:

```
project Test_Baseline extends "Baseline" is
   for Source_Dirs use ( "test_code" );
   for Main use ( "test_driver" );
end Test_Baseline;
```

# Lab

# Structuring Your Application Lab

- Source is included in folder `gpr_3_structuring_your_application`
- **Very** simplistic speed monitor
  - Reads current distance
  - Determines amount of time since last read
  - Calculates speed
  - Sends message
- Four subsystems
  - `Base` - types and speed calculator
  - `Sensors` - reads distance from some register
  - `Messages` - sends message to some memory location
  - `Application` - main program
- We could build one GPR file and point to all source directories
  - But as our application grew, this would become harder to maintain

# Assignment Part One

1. Build GPR files for each subsystem
   - Hint: These subsystems *depend* on each other, they do not *override* source files
   - As you build each GPR file, run `gprbuild -P <gprfile>` to make sure everything works
   - Main program is in `main.adb`
2. Run `main`
   - This will fail (leading up to Part Two of the assignment)
3. Modify `base_types.ads`
   - Just so source code needs to be compiled
4. Rebuild your main program
   - Even though the modified source file is not directly referenced in the main GPR file, GPRBUILD should compile everything it needs

# Assignment Part One - Solution

```
with "../base/base.gpr";
with "../messages/messages.gpr";
with "../sensors/sensors.gpr";
project Application is
   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("main.adb") & project'Main;
end Application;

with "../base/base.gpr";
project Messages is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Messages;

with "../base/base.gpr";
project Sensors is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Sensors;

project Base is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Base;
```

# Assignment Part Two

1. Build GPR files to create test stubs for `Odometer` and `Sender`
   - Test bodies exist in the appropriate `test` subfolders
   - Create extensions for `messages.gpr` and `sensors.gpr`
     - We want to inherit the package spec, but use the "test" package bodies
2. Build a GPR file for the main application
   - `Main` still works, we just need the GPR file to access our stubs
   - We could create a new GPR file, or extend the original. Which is easier?
3. Build and run your main program

# Assignment Part Two - Solution

- `messages/test` directory

```
project Messages_Test extends "../Messages.gpr" is
   for Source_Dirs use (".");
end Messages_Test;
```

- `sensors/test` directory

```
project Sensors_Test extends "../sensors.gpr" is
   for Source_Dirs use (".");
end Sensors_Test;
```

- `test` directory

```
with "../messages/test/messages_test.gpr";
with "../sensors/test/sensors_test.gpr";
project Test extends "../application/application.gpr" is
   for Main use ("main.adb") & project'Main;
end Test;
```

# Advanced Capabilities

# Introduction

# Other Types of GPR Files

- Project files can also be used for
  - Building libraries
  - Building systems
- Project files can also have children
  - Similar to Ada packages

# Library Projects

# Libraries

- Subsystems packaged in specific way
- Represented by project files with specific attributes
- Referenced by other project files, as usual
  - Contents become available automatically, etc.
- Library Project

```
library project Static_Lib is
   -- keyword "library" is optional
   ...
end Static_Lib;
```

- Standard Project referencing library

```
with "static_lib.gpr";
project Main is
   ...
end Main;
```

# Creating Library Projects

- Several global attributes are involved/possible
- Required attributes

Library_Name Name of library

Library_Dir Where library is installed

- Important optional attributes

Library_Kind *static*, *static-pic*, *dynamic*, *relocatable* (same as *dynamic*)

Library_Interface Restrict interface to subset of units

Library_Auto_Init Should autoinit at load (if supported)

Library_Options Extra arguments to pass to linker

Library_GCC Use custom linker

# Supported Library Types

- Static Libraries
  - Code statically linked into client applications
  - Becomes permanent part of client during build
  - Each client gets separate, independent copy
- Dynamic Libraries
  - Code dynamically linked at run-time
  - Not permanent part of application
  - Code shared among all clients
- Stand-Alone Libraries (SAL)
  - Minimize client recompilations when library internals change
  - Contain all necessary elaboration code for Ada units within
  - Can be static or shared
- See the *GNAT Pro Users Guide* for details

# Static Library Project Example

```
library project Name is
   for Source_Dirs use ("src1", "src2");
   for Library_Dir use "lib";
   for Library_Name use "name";
   for Library_Kind use "static";
end Name;
```

- Creates library `libname.a` on Windows

# Standalone Library Project Example

```
library project Name is
   Version := "1";
   for Library_Interface use ("int1", "int1.child");
   for Library_Dir use "lib";
   for Library_Name use "name";
   for Library_Kind use "relocatable";
   for Library_Version use "libdummy.so." & Version;
end Name;
```

- Creates library `libname.so.1` with a symlink `libname.so` that points to it

# Aggregate Projects

# Complex Applications

- Many applications have multiple exectuables and/or libraries
  - Shared source code
  - Multiple "top-level" project files
- Assume project A `withs` project B and project C
  - Build of project A will only compile/link whatever is necessary for project A's executable(s)
  - Executables in project B and C will need to be generated separately
  - Running `gprbuild` on all three projects causes redundant processing
    - Determination of files that need to be compiled
    - Libraries are always built when `gprbuild` is called

# Aggregate Projects

- Represent multiple, related projects
  - Related especially by common source code
- Allow managing options in a centralized way
- Compilation optimized for sources common to multiple projects
  - Doesn't compile more than necessary

# Aggregate Project Example

```
aggregate project Agg is
   -- Projects to be built
   for Project_Files use ("A.gpr", "B.gpr", "C.gpr");
   -- Directories to search for project files
   for Project_Path use ("../dir1", "../dir1/dir2");
   -- Scenario variable
   for external ("BUILD") use "PRODUCTION";

   -- Common build switches
   package Builder is
      for Global_Compilation_Switches ("Ada")
            use ("-O1", "-g");
   end Builder;
end Agg;
```

# Child Projects

# Grouping Projects

- Sometimes we want to emphasize project relationships
  - Similar to parent/child relationship in Ada packages
- Child project
  - Declare child of project same as in Ada:
    `project Parent.Child ...`
  - **No inheritance assumed** (unlike Ada)
  - Behavior of child follows normal project definition rules

# Child Projects

- Original project

```
-- math_proj.gpr
project Math_Proj is
   ...
end Math_Proj;
```

- Child *depends* on parent

```
with "math_proj.gpr";
project Math_Proj.Tests is
   ...
end Math_Proj.Tests;
```

- Child *extends* parent

```
project Math_Proj.High_Performance extends "math_proj.gpr" is
   ...
end Math_Proj.High_Performance;
```

- Illegal project

```
project Math_Proj.Test is
   ...
end Math_Proj.Test;
```

# Summary

# Conclusion

# GNAT Project Manager Summary

- Supports hierarchical, localized build decisions
- IDEs provide direct support
- GPRBUILD allows broad or narrow control over build process
- See the *GPRbuild and GPR Companion Tools User's Guide* for further functionality and capabilities
  - Target build processing
  - Distributed builds
  - Etc

# Day 4 - PM

# GNATstub

# Introduction

# Body Stub Generator

- Creates empty (but compilable) package/subprogram bodies
- Can use GNAT Project file
  - Configuration in package `gnatstub`
- Default behavior is to raise exception if stub is called
  - It means you did not create a "real" body

# Why Do You Need Stubs?

Sometimes we want to establish code structure quickly

- Start prototyping code architecture first
- Worry about implementation details later
    - Don't want to get caught in compilation details/behavior in early development

# Running GNATstub

# Running GNATSTUB

```
gnatstub [switches] filename
```

where `filename` can be a package spec or body

- Package spec
  - GNATSTUB will generate a package body containing "dummy" bodies for subprograms defined but not completed in the spec
- Package body
  - For any subprogram defined as `separate` in the package body, a file will be created containing a body for the subprogram

**Note**

Need to specify `--subunits` switch

# Example Package Spec

- Filename `example.ads` contains

```
package Example is
   procedure Null_Procedure is null;
   procedure Needs_A_Stub;
   function Expression_Function return Integer is (1);
end Example;
```

- `gnatstub example.ads` will generate `example.adb`

```
pragma Ada_2012;
package body Example is

   ------------------
   -- Needs_A_Stub --
   ------------------

   procedure Needs_A_Stub is
   begin
      pragma Compile_Time_Warning
        (Standard.True, "Needs_A_Stub unimplemented");
      raise Program_Error with "Unimplemented procedure Needs_A_Stub";
   end Needs_A_Stub;

end Example;
```

# Example Package Body

- Filename `example.adb` contains

```
package body Example is
   procedure Do_Something_Else;
   procedure Do_Something is separate;
   procedure Do_Something_Else is
   begin
      Do_Something;
   end Do_Something_Else;
end Example;
```

- `gnatstub --subunits example.adb` will generate `example-do_something.adb`

```
pragma Ada_2012;
separate (Example)
procedure Do_Something is
begin
   pragma Compile_Time_Warning (Standard.True, "Do_Something unimplemented");
   raise Program_Error with "Unimplemented procedure Do_Something";
end Do_Something;
```

# GNATstub Switches

# Controlling Behavior When Called

- By default, a stubbed subprogram will raise `Program_Error` when called
    - Procedures use a `raise` statement
    - Functions use a `raise` expression in a `return`
        - To prevent warnings about no return in a function
- You can disable the exception in procedures
    - Switch `--no-exception`

**⚠ Warning**

Functions still need a return statement, so `raise` expression is still present

# Formatting Comment Blocks

- Sometimes you use GNATSTUB to create a shell for your implementation
  - Having the tool populate the shell with comments can be helpful
- Comment switches:

`--comment-header-sample`
*Create a file header comment block*

`--comment-header-spec`
*Copy file header from spec into body*

`--header-file=<filename>`
*Insert the contents of* `<filename>` *at the beginning of the stub body*

- Default behavior is to add a comment block for each subprogram
  - Use `--no-local-header` to disable this

# Other Common Switches

`files=<filename>`

`<filename>` contains a list of files for which stubs will be generated

`--force`

Overwrite any existing file (without this, GNATSTUB will flag as an error)

`--output-dir=<directory>`

Put generated files in `<directory>`

`max-line-length=<nnn>`

Maximum length of line in generated body. Default is 79, maximum is 32767

# Lab

# GNATSTUB Lab

- We are going to implement a simple math package that does addition and subtraction
  - The exectuable takes 3 numbers on the command line - adds the first two, subtracts the third, and prints the result
- Copy the `gnatstub` lab folder from the course materials location `source` folder
- Contents of the folder:
  - `default.gpr` - project file
  - `main.adb` - main program
  - `math.ads` - package spec that we want to implement

**Note**

We use animation - if you don't know the answer, Page Down should give it to you

# Build the Executable

1. Open a command prompt window and navigate to the directory containing `default.gpr`
2. Try to build the exectuable (`gprbuild -P default.gpr`)
   - Build fails because `Math` is not implemented
3. Build a stub for `Math`
   - Make sure you copy the file header comment into the stub

# Build the Executable

1. Open a command prompt window and navigate to the directory containing `default.gpr`
2. Try to build the exectuable (`gprbuild -P default.gpr`)
    - Build fails because `Math` is not implemented
3. Build a stub for `Math`
    - Make sure you copy the file header comment into the stub

```
gnatstub --comment-header-spec math.ads
```

# Build the Executable Again

1. Build the executable again
   - Builds, but you get compile warnings from the stubbed subprograms
2. Run the executable
   - Remember to add three numbers on the command line
3. Executable should fail with `Program_Error` in `Add_Two_Numbers`
   - Default stub behavior
4. Rebuild the stub without exceptions and run it again

# Build the Executable Again

1. Build the executable again
   - Builds, but you get compile warnings from the stubbed subprograms
2. Run the executable
   - Remember to add three numbers on the command line
3. Executable should fail with `Program_Error` in `Add_Two_Numbers`
   - Default stub behavior
4. Rebuild the stub without exceptions and run it again

```
gnatstub -f --comment-header-spec --no-exception math.ads
```

- Exception now raised in `Subtract_Two_Numbers`
  - Exceptions always raised for functions in a stub

# Implement `Math`

1. Edit the `Math` package body to implement the two subprograms
2. Build and run the executable

# Math Package Body

```
----------------------------------------------------------
--                                                      --
-- MATH                                                 --
--                                                      --
--                                                      --
-- Simplistic math package to add or subtract two numbers --
--                                                      --
----------------------------------------------------------

pragma Ada_2012;
package body Math is

   ---------------------
   -- Add_Two_Numbers --
   ---------------------

   procedure Add_Two_Numbers
     (Result : out Integer; Param_A : Integer; Param_B : Integer)
   is
   begin
      Result := Param_A + Param_B;
   end Add_Two_Numbers;

   --------------------------
   -- Subtract_Two_Numbers --
   --------------------------

   function Subtract_Two_Numbers
     (Param_A : Integer; Param_B : Integer) return Integer
   is
   begin
      return Param_A - Param_B;
   end Subtract_Two_Numbers;

end Math;
```

# Summary

# Improving on GNATSTUB

- Sometimes empty code stubs aren't enough
  - Not only don't they do anything useful, they actively raise compiler warnings and run-time exceptions!
- "Smart" stubs are useful for testing
  - Replace code not available for testing
  - Control/replace external interfaces when testing natively
    - Read sensors
    - Write to a console
- You can modify the generated stub(s) to implement all this

# Beyond GNATSTUB

- User-created "Smart" stubs are great for testing
  - But there's a lot of repetition in building the stubs
  - And maintenance can be difficult
- Use GNATTEST to create more advanced unit tests
  - Expands on stubbing capabilities
  - Adds test driver generation
  - Adds automation capabilities

For more information, go to GNATtest (https://www.adacore.com/dynamic-analysis/gnattest)

# GNATSTACK

# Introduction

# Determining Maximum Stack Size

- GNATSTACK statically computes maximum stack space
  - For each application entry point (including tasks)
- **Static** Analysis
  - Analyzes artifacts of compiler (not run-time execution)
  - Computes worst-case stack requirements
  - When no assumptions made, stack size never exceeds analyzed value

# Outputs

- Worst-case stack requirements for each entry point
    - Entry points can be deduced from source or specified by user
- Path leading to each scenario
- *Visualization of Compiler Graphs (VCG)*
    - File containing complete call tree for application
    - Contains both local and accumulated stack usage

# Optional Analysis Outputs

- **Indirect calls** (including dispatching)
  - Number of indirect calls from any subprogram
- **External calls**
  - All subprograms reachable from entry point where no stack/call graph information is available
- **Unbounded frames**
  - All subprograms reachable from entry point with unbounded stack requirements
  - Stack size depends on arguments passed to the subprogram
- **Cycles**
  - Detect call cycles in the call graph
  - Represent potential recursion for possibly unbounded stack consumption

# Running GNATstack

# Example Subprogram

```
procedure Main_Unit is
   type Data_Type is array (1 .. 5) of Integer;

   function Inverse (Input : Data_Type) return Data_Type is
      Result : Data_Type;
   begin
      for Index in Data_Type'Range loop
         Result (Index) := Input (Data_Type'Last -
                                  (Index - Data_Type'First));
      end loop;

      return Result;
   end Inverse;

   Data   : Data_Type := (1, 2, 3, 4, 5);
   Result : Data_Type;
begin
   Result := Inverse (Data);
end Main_Unit;
```

# Getting Started with GNATSTACK

Two parts of performing stack analysis

1. Generation of stack consumption and call-graph information

   ```
   gprbuild --RTS=light main_unit.adb -cargs -fcallgraph-info=su
   ```

   *We use the light runtime to avoid including things like the secondary stack*

2. Analysis and report generation

   ```
   gnatstack *.ci
   ```

Which generates the following report:

```
Worst case analysis is *not* accurate because of external calls. Use -Wa for details.

Accumulated stack usage information for entry points

main : total 224 bytes
 +-> main
 +-> main_unit
 +-> main_unit.inverse
```

*Note that the actual stack usage can depend on things like runtime, operating system, and compiler version.*

# GNATstack Switches

# Execution-Related Switches

`-e main1[,main2[,...]]` → Use list of subprograms as entry points

`-a` → Use all subprograms as entry points

`-f filename` → Store callgraph in `filename`

- If not specified, stored in `graph.vcg`

`-P project` → Use GPR file `project` to find `*.ci` files

# Commonly Used Switches

`-v` → verbose

- Show source location for subprogam

`-o={a,s}` → order for displaying call graphs

- **a** sort alphabetically
- **s** sort by stack usage (default)

`-t={i,d,a}` - print target for indirect/dispatching calls

- **i** for indirect calls only
- **d** for dispatching calls only
- **a** for both indirect and dispatching calls

# Lab

# GNATstack Lab

- We are going to perform stack analysis on some source code examples
  - Although this is called a lab, it's more like a walk-through!
- Copy the `gnatstack` lab folder from the course materials location `source` folder
- Contents of the folder:
  - `simple` - folder containing a simple main procedure
  - `complicated` - folder containing multiple main procedures and some other packages

**Note**

We use animation - if you don't know the answer, Page Down should give it to you

# Getting Familiar with GNATSTACK

1. Open a command prompt window and navigate into the folder `simple`
2. Build the executable `main_unit`, making sure to generate call-graph information
   - Don't forget to use the light-tasking runtime (switch `--RTS=light`)
3. Perform the stack analysis

# Getting Familiar with GNATstack

1. Open a command prompt window and navigate into the folder `simple`
2. Build the executable `main_unit`, making sure to generate call-graph information
   - Don't forget to use the light-tasking runtime (switch `--RTS=light`)

```
gprbuild --RTS=light main_unit.adb -cargs -fcallgraph-info=su
```

3. Perform the stack analysis

# Getting Familiar with GNATSTACK

1. Open a command prompt window and navigate into the folder `simple`
2. Build the executable `main_unit`, making sure to generate call-graph information
   - Don't forget to use the light-tasking runtime (switch `--RTS=light`)

```
gprbuild --RTS=light main_unit.adb -cargs -fcallgraph-info=su
```

3. Perform the stack analysis

```
gnatstack *.ci
```

```
main : total 224 bytes
 +-> main
 +-> main_unit
 +-> main_unit.inverse
```

The numbers may be different, but the calls should match

# Adding Source Information

To see where the total number of bytes comes from, run the analysis in *verbose* mode

# Adding Source Information

To see where the total number of bytes comes from, run the analysis in *verbose* mode

```
gnatstack -v *.ci
```

# Verbose Mode Output

Verbose mode also shows full path to the source

```
+-> main at main:b__main_unit.adb:20:4 : 64 bytes
+-> main_unit at Main_Unit:L:\\main_unit.adb:1:1,
    ada_main_program:b__main_unit.adb:17:14 : 96 bytes
+-> main_unit.inverse at Inverse:L:\\main_unit.adb:4:4 : 64 bytes
```

# Working with Multiple Mains

1. Open a command prompt window and navigate into the folder `complicated`
2. Examine the GNAT Project file and notice the following:
   - The `-fcallgraph-info=su` switch is specified in the `Compiler` package
   - All main subprograms are specified using `for Main`
     - Otherwise `gprbuild` does not know what executables to build
   - The runtime is specified using `for Runtime`
3. Build all the executables using the included `default.gpr`

```
gprbuild -P default.gpr
```

# Recursive Calls (Cycles)

```
procedure Odd (Number : in out Integer) is
begin
   Number := Number - 1;
   if Number > 0 then
      Cycles (Number);
   end if;
end Odd;

procedure Even (Number : in out Integer) is
begin
   Number := Number - 2;
   if Number > 0 then
      Cycles (Number);
   end if;
end Even;

procedure Cycles (Number : in out Integer) is
   Half : constant Integer := Number / 2;
begin
   if Half * 2 = Number then
      Even (Number);
   else
      Odd (Number);
   end if;
end Cycles;
```

# Investigating Cycles

1. Perform the stack analysis for `Cycles_Main`

# Investigating Cycles

1. Perform the stack analysis for `Cycles_Main`

```
gnatstack -e cycles_main *.ci
```

```
Worst case analysis is *not* accurate because of cycles, external calls. Use -Wa for details.

Accumulated stack usage information for entry points

cycles_main : total 176+? bytes
 +-> cycles_main
 +-> cycles_example.cycles *
 +-> cycles_example.odd *
 +-> <__gnat_last_chance_handler> *
```

2. Notice the warning indicating to use `-Wa` for details - try that.

# Investigating Cycles

1. Perform the stack analysis for `Cycles_Main`

```
gnatstack -e cycles_main *.ci
```

```
Worst case analysis is *not* accurate because of cycles, external calls. Use -Wa for details.

Accumulated stack usage information for entry points

cycles_main : total 176+? bytes
 +-> cycles_main
 +-> cycles_example.cycles *
 +-> cycles_example.odd *
 +-> <__gnat_last_chance_handler> *
```

2. Notice the warning indicating to use `-Wa` for details - try that.

```
gnatstack -Wa -e cycles_main *.ci
```

Notice the added information

```
List of reachable cycles:

<c1> cycles_example.cycles
 +-> cycles_example.cycles
 +-> cycles_example.even
 +-> cycles_example.cycles

<c2> cycles_example.cycles
 +-> cycles_example.cycles
 +-> cycles_example.odd
 +-> cycles_example.cycles
```

# Subprogram Pointers (Indirect Calls)

```
type Subprogram_Access_T is access procedure
     (A, B :     Integer;
      C    : out Boolean);
procedure Procedure_One
  (A, B :     Integer;
   C    : out Boolean) is
begin
   C := A > B;
end Procedure_One;

procedure Procedure_Two
  (A, B :     Integer;
   C    : out Boolean) is
begin
   C := A < B;
end Procedure_Two;

Calls : array (Boolean) of Subprogram_Access_T :=
  (Procedure_One'Access,
   Procedure_Two'Access);

procedure Test (Flag : in out Boolean) is
begin
   Calls (Flag).all (1, 2, Flag);
end Test;
```

# Investigating Indirect Calls

1. Perform the stack analysis for `Indirect_Main`

# Investigating Indirect Calls

1. Perform the stack analysis for `Indirect_Main`

```
gnatstack -e indirect_main *.ci
```

```
Worst case analysis is not accurate because of external calls, indirect calls. Use -Wa for details.

Accumulated stack usage information for entry points

indirect_main : total 112+? bytes
 +-> indirect_main
 +-> indirect_example.test
 +-> indirect call *
```

2. Notice the warning indicating to use `-Wa` for details - try that.

# Investigating Indirect Calls

1. Perform the stack analysis for `Indirect_Main`

```
gnatstack -e indirect_main *.ci
```

```
Worst case analysis is not accurate because of external calls, indirect calls. Use -Wa for details.

Accumulated stack usage information for entry points

indirect_main : total 112+? bytes
 +-> indirect_main
 +-> indirect_example.test
 +-> indirect call *
```

2. Notice the warning indicating to use `-Wa` for details - try that.

```
gnatstack -Wa -e indirect_main *.ci
```

Notice the added information

```
List of reachable external subprograms:

  <__gnat_last_chance_handler>

List of reachable and unresolved indirect (including dispatching) calls:

  1 indirect call in: indirect_example.test
    at L:\indirect_example.adb:26
```

# Using Other Switches

If you have time, experiment with some other switches

- Show information for multiple main programs

- Show target for dispatching calls

# Using Other Switches

If you have time, experiment with some other switches

- Show information for multiple main programs

```
gnatstack -e indirect_main,cycles_main *.ci
```

- Show target for dispatching calls

# Using Other Switches

If you have time, experiment with some other switches

- Show information for multiple main programs

```
gnatstack -e indirect_main,cycles_main *.ci
```

- Show target for dispatching calls

```
gnatstack -td -e dispatching_main *.ci
```

# Summary

# Improving on GNATSTACK

- When static analysis doesn't have enough information, user can provide via switches
  - `-c <size>` - use **size** number of bytes for cycle entry
  - `-d <size>` - use **size** number of bytes for dynamic (unbounded) calls
  - `-u <size>` - use **size** number of bytes for external (unknown) calls
- Limitations due to call stack begin non-deterministic
  - Recursive calls - how deep is the recursion?
  - Indirect calls - subprogram pointers
  - Dispatching calls - subprogram dispatching

# Beyond GNATstack

## GNAT Static Analysis Suite (SAS)

- In-depth static analysis tool
- Defect and vulnerability analysis
- Code metrics
- Coding standards verification