Ada Declarations

Barnes, chapter 5

Identifiers

- **Ada identifiers are case insensitive**
  - HELLO = hello = HellO
- **Start with a letter**
- **Ends with a letter or a number**
- **May contain non-consecutive underscores**

Which of the following are legal?

- Something
- My__Id
- _Hello
- A_67_9
- _CONSTANT
- 09_A_2
- YOP_

Comments

- **Ada provides end of line comments with --**

`--  This is an Ada comment`	`// This is a C++ comment`

- **There is no block comment (/* */)**

Numbers

- **The underscore is allowed for numbers**
  - 1_000_000 = 1000000
- **Numbers can be expressed with an integer base (from 2 to 16)**
  - 10#255# = 2#1111_1111# = 8#377# = 16#FF#
- **Numbers can be defined with an exponent part**
  - 2#1#E8 = 256
  - 2E8 = 200000000
- **Real literals must have a dot**
  - With a digit before and after the dot.
- **All of this can be combined and works for real literals as well**
  - 2#11.1#E2 = 14.0
- **Exponent is always a base-10 integer**

Variables declarations

- **Defined by one (or several) names, followed by :, followed by type reference and possibly an initial value**

```ada
A : Integer;
B : Integer := 5;
C : constant Integer := 78;
D, E : Integer := F (5);
```

```c
int A;
int B = 5;
const int C = 78;
int d = F (5), e = F(5);
```

- **Elaboration is done sequentially**

```ada
A : Integer := 5;
B : Integer := A;
C : Integer := D; -- COMPILATION ERROR
D : Integer := 0;
```

- **Initialization is called for each variable individually**

```ada
A, B : Float := Compute_New_Random;
--  This is equivalent to:
A : Float := Compute_New_Random;
B : Float := Compute_New_Random;
```

- **":=" on a declaration is an initialization, not an assignment (special properties, mentioned later)**

Numeric values

- **No need to give the size – deduced from the context**

Ada	C
`A : Long_Integer := 0;`	`long int A = 0L;`

- **It's possible to declare "named numbers" with infinite precision**

```ada
NN : constant := 1.0 / 3.0;

X  : Float := NN;
X2 : Long_Float := NN;
X3 : Long_Long_Float := NN;
-- equals 3.33333333333333333E-01

X4 : Long_Long_Float := X;
-- equals 3.33333343267440796E-01
```

```c
#define NN 1.0 / 3.0

float X = NN;
long float X2 = NN;
long long float X3 = NN;


long long float X4 = X;
```

Declarative blocks

- **Declarations can only occur in declarative parts**
- **Statements can only occur in the statement parts**
- **Sub-declaration blocks can be introduced with a *block statement***

<pre>**declare**
   A : Integer := 0;
begin
   A := A + 1;
end;</pre>
	<pre>{
	   int A = 0;
	
	   A++;
	}</pre>

Scope

- **Defines a declaration lifetime**
- **The scope from an object goes from its declaration point to the corresponding "end"**

```ada
declare
   A : Integer;
begin
   -- code
   declare
      B : Integer;
   begin
      -- code
   end;
   A := B; -- COMPILATION ERROR
end;
```

```c
{
   int A;

   // code
   {
      int B;

      // code
   }
   A = B;
}
```

Visibility

- **Nested scopes can "hide" declarations from outer scopes**

```ada
declare
   A : Integer;
begin
   -- references to the outer A
   declare
      A : Float;
   begin
      -- references to the inner A
   end;
 end;
```

```c
{
   int A;

   // references to the outer A
   {
      float A;

      // references to the inner A
   }
   A = B;
}
```

- **With named scopes, it's still possible to have access to outer entities**

```ada
Outer : declare
   A : Integer;
begin
   declare
      A : Float;
      B : Integer;
   begin
      A := Outer.A;
```

Some Terminology...

- **In a block statement, or subprogram body:**

```
declare
   -- "Declarative part"
   subtype S is Integer range 0 .. 10; -- a declaration
   A : S; --  another declaration
begin
   -- "Statement Part"
   S1; -- A statement
   S2; -- Another statement

   A := X + Y; -- An assignment statement containing
               -- a Name (left hand side) and
               -- an Expression (right hand side).
end;
```

Some Terminology...

- **Statements are *executed*.**
- **Expressions are *evaluated*.**
- **Declarations are *elaborated*.**

- **A *Static* Expression is evaluated at compile-time.**
- **A *Dynamic* Expression is evaluated when the program is running.**

- **Note for C and C++ users: expressions and statements are completely separate things in Ada, and are not interchangeable...**

Quiz

Is there a compilation error? (1/10)

```
V : Natural := 7;
J : constant Natural := V + 4;
```

Is there a compilation error? (2/10)

```
V : Natural := 7;
V : Real := 5.5;
```

Is there a compilation error? (3/10)

```
V : Natural := 7;
V : Natural := V + 5;
```

Is there a compilation error? (4/10)

```
V : Natural := V * 0;
```

Is there a compilation error? (5/10)

```
V : Natural := 5;
declare
   V : Natural := V * 2;
```

Is there a compilation error? (6/10)

```
V : Float := 5.0;
```

Is there a compilation error? (7/10)

```
V : Float := 5.;
```

Is there a compilation error? (8/10)

```
ClassRoom : constant Natural := 5;
Next_ClassRoom : Natural := classroom + 1;
```

Is there a compilation error? (9/10)

```
Class__Room : constant Natural := 5;
```

Is there a compilation error? (10/10)

```
_my_value : constant Natural := 5;
```

Ada Basic Types

Barnes, chapter 6

Ada Strong Typing

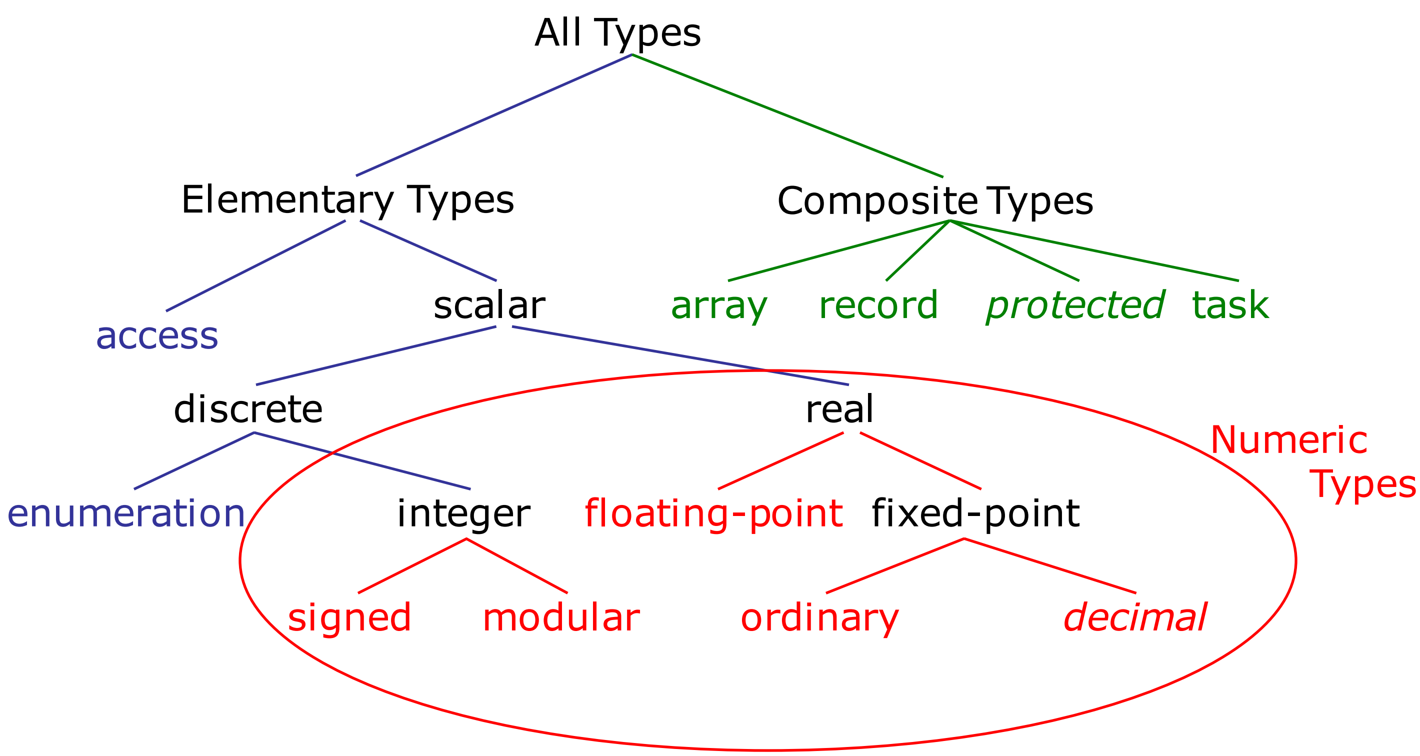
- **Types are at the base of the Ada model**
- **Semantics ≠ Representation**
- **All Ada types are named**
  - (Well, almost all)
- **Associated with properties (ranges, attributes…) and operators**

```
A : Integer := 10 * Integer (0.9);
A : Integer := Integer
   (Float (10) * 0.9);
```

```
int A = 10 * 0.9
```

- **The compiler will warn in case of inconsistencies**

Types hierarchy



Defining a Type

- **New types can be created in declaration scopes**

```
type <name> is <definition> [with predicate];
type <name> is new <definition> [with predicate];
```

- **Discrete types**

```
type Score is range 0 .. 20;
type Color is (Red, Blue, Green);
type Oranges is new Positive;
type Apples is new Positive;
type Byte is mod 2**8;
```

- **Floating point types**

```
type Size is new Float;
type Low_Precision is digits 4;
```

- **Fixed point types**

```
type Cm is delta 0.125 range 0.0 .. 240.0;
type Euro is delta 0.01 digits 15;
```

What’s an enumeration (for a C programmer)

- **An enumerated type is a scalar type**
  - Finite set of values
  - Ordered

Type name

Values

```
type Color is (Red, Blue, Green);
```

- **Each value has a name**
  - Either an identifier
  - Or a character
- **No relationship with integer**
- **Boolean is an enumerated type**

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

Integer types

- **Signed integer types are defined by a range**

Values

Type name

```
type Columns is range 1 .. 80;
```

  - No values outside the range

- **Modular type are defined by a modulus**

Modulus

Type name

```
Type Byte is mod 256;
```

  - Wrap-around semantic of operators

Floating point types

- **Defined by relative precision**
  - Minimum number of significant decimal digits

precision

Type name

```
type Real is digits 8;
```

  - May have a range

Type name

```
type Real is digits 8 range 0.0 .. 1.0E10;
```

Type Attributes

- **Accessed through '**

```
T'First           -- first value of the type
T'Last            -- last value of the type
T'Range           -- equivalent to T’First .. T’Last
T'Succ (V)        -- return the next value in the order
T’Pred (V)        -- return the previous value in the order
T'Image (V)       -- return a string representation of the value
T'Value (S)       -- converts to a value representation
T'Pos (V)         -- Return a position based on a value
T'Val (I)         -- Return a value based on a position
T'Min (V1, V2)    -- Return the min between two values
T'Max (V1, V2)    -- Return the max between two values
T'Ceiling (V)     -- Returns the smallest integral value after V
T'Floor (V)       -- Returns the largest integral value before V
T'Truncation (V)  -- Truncates the value towards 0
T'Size            -- Return the size of the values of the type
T'Rounding (V)    -- Rounds to the closest integer
```

- **Example**

```
V : Character := Character'Val (0);
S : String := Integer'Image (42);
```

Subtypes

- **Subtypes add a constraint to a type**

```
subtype D is Integer range 0 .. 9;
```

- **Subtypes do not create new types, and do not require type conversion**

```
   subtype D is Integer range 0 .. 9;
   A : Integer := 0;
   B : D := 1;
begin
   A := A + B;
```

- **The language offers some basic subtypes**

```
subtype Positive is Integer range 1 .. Integer'Last;
subtype Natural  is Integer range 0 .. Integer'Last;
```

Base Type

- **T'Base is the type used by the compiler to implement the type according to the constraints**

```
type Small_Int is range 0 .. 10;
```

Fits in a 8-bits integer

- **Base types can be used for overflow checks (see later)**

- **Base types can be used as a regular type**

```
Put_Line (Small_Int'Base'Image (Small_Int'Base'First);
--  => -128 (implementation-dependent)
Put_Line (Small_Int'Base'Image (Small_Int'Base'Last);
--  => 127 (implementation-dependent)
```

Subtype checks / Overflow checks

- **Types and subtypes can be associated with subtype checks**

Valid values are between 0 and 10

```
type Small_Int is range 0 .. 10;
```

- **Subtype checks are computed in well defined places (assignment, parameter passing and conversions…)**

```
V1 : Small_Int := 11; -- Exception
```

- **In expressions, overflow checks are performed on intermediate values:**

```
V1 : Small_Int := 2; -- OK
V2 : Small_Int := V1 + 10 – V1; -- OK, equals 10
V3 : Small_Int := (V1'Base'Last + 1) / 100; -- NOK, overflow check
```

Dynamic Expression vs. Static Expression

- **Ada differentiates static expressions and dynamic expressions**

- **Static expressions are expressions including**
  - literals
  - calls to static predefined functions and attributes
  - constants initialized with static expressions

- **Static expressions are evaluated at compile-time**

- **Static expressions are required by some constructs**

Constants and Named Numbers

- It is possible to create a constant value

```
C : constant Integer := 0;
```

- A *constant* inherits from all properties of its types, except that it can't be written. In particular, it has to respect boundaries.
- A constant can be initialized through a *dynamic* expression, but is then *read-only* for its lifetime.

- A *named number* doesn't have a type
- It must be valuated by a static expression
- It can represent data out of bounds

```
N : constant := 2 ** 128;
```

- Exceptions can be raised at run-time when used

```
V1 : Integer := N - N + 1; -- OK
V2 : Integer := N;         -- NOK
```

Conversion / Qualification

- **In certain cases, types can be converted from one to the other**
  - They're of the same structure (e.g. Numeric)
  - One is the derivation of the other
- **Conversion needs to be explicit**

```
V1 : Float := 0.0;
V2 : Integer := Integer (V1);
```

- **A qualification can be used to specify the type or subtype of an object - it doesn't convert it**

```
V1 : Integer := 0;
V2 : Integer := Natural'(V1);
```

- **Qualification is most useful when fixing ambiguities (see later)**

Quiz

Is there a compilation error? (1/10)

```
V : Float := 10;
```

What's the output of this code? (2/10)

```
    type Float_1 is digits 5;
    type Float_2 is digits 7;

    V_1 : Float_1 := 10.0E10;
    W_1 : Float_1 := V_1 + 1.0;

    V_2 : Float_2 := 10.0E10;
    W_2 : Float_2 := V_2 + 1.0;
begin
    Put_Line (Boolean'Image (V_1 = W_1));
    Put_Line (Boolean'Image (V_2 = W_2));
```

Is there an error? (3/10)

```
type X is mod 10;

V1 : X := 10;
V2 : X := 9 + 1;
```

What’s the output of this code? (4/10)

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

Is there an exception? (5/10)

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

   V : T := 9;
   W : T := 2;
begin
   V := V + W - 1;
```

Is there an exception? (6/10)

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

   V : T := 9;
   W : T := 2;
begin
   V := T (V + W) - 1;
```

Is there a compilation or runtime error? (7/10)

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

Is there a compilation error? (8/10)

```
type T is (A, B, C);

V1 : T := T'Val ("A");
V2 : T := T'Value (2);
```

Is there a run-time error? (9/10)

```
type T is (A, B, C);

V1 : T := T'Value ("A");
V2 : T := T'Value ("a");
V3 : T := T'Value (" a ");
```

Is there a compilation error? (10/10)

```
type T is range 1 .. 0;
V : T;
```

Quiz #2

What is a type ?

What is a type ?

- A (finite) set of values
- Operations on this set
- Physical representation

Ada vs. C typing

- **In Ada, you can create new types for every kind of type**
  - Including integers, unsigned
- **Strong typing**
- **(Almost) no built-in types**
  - Except Boolean
  - You don't need to use predefined types
- **You can create new operators**
- **You can specify physical representation**

Statements

Barnes chapter 7

Simple statements

- **The main Simple Statements**
  - Null
  - Assignment
  - *Procedure Call* and *Return* will be dealt with when we get to Subprograms
  - *Raise Statement* will be covered under Exceptions.
  - *Exit Statement* will be covered with Loops.
  - The rest are to do with *Tasking*.

Null statement

- **The Null Statement in Ada is written explicitly:**

```
null;
```

- **This was a deliberate design decision in Ada to make it very hard to "accidentally" write a null statement.**
- **Compare:**

`for I in 1 .. 10 loop`
`   null;`
`end loop;`
	`for (i = 1; i <= 10; i++);`

Assignment statement

- **Very simple syntax:**

```
variable_name := expression ;
```

- **A “name” in Ada can be “dotted” to include package names and record components, and also contain parentheses for array elements and so on.**
- **For example:**

```
P.State (1).F1 := 6;
```

Compound statements

- **In Ada, statements are *terminated* with a semicolon ‘;’**
- **The main compound statements**
  - If
  - Case
  - Loop
  - Block

  - The remainder are concerned with *Tasking*

If Statements

- **If statements**

```ada
if A = 0 then
   Put_Line ("A is 0");
elsif B = 0 then
   Put_Line ("B is 0");
else
   Put_Line ("Else...");
end if;
```

```c
if (A == 0) {
   printf ("A is 0");
} else if (B == 0) {
   printf ("B is 0");
} else {
   printf ("Else...");
}
```

Condition symbols

- **Comparison**
  - /=
  - =
  - >=
  - <=
  - >
  - <
- **Boolean operators**
  - and
  - or
  - xor
  - and then
  - or else
  - not (unary)

Beware of the boolean operators

- “and”, “or” are not short-circuit, both operands are always evaluated

```
if X /= 0 and Y / X > 1 then -- MAY RAISE AN EXCEPTION
```

- The short-circuit operators are “and then” and “or else”

```
if X /= 0 and then Y / X > 1 then -- OK
```

Case Statement

```
case A is
   when 0 =>
      Put_Line ("zero");

   when -9 .. -1 | 1 .. 9 =>
      Put_Line ("digit");
   when others =>
      Put_Line ("other")
end case;
```

No fall through

```
switch (A) {
   case 0:
      printf ("0");
      break;
   case -9:case -8:case -7:case -6:
   case -5:case -4:case -3:case -2:
   case -1:case 1:case 2:case 3:
   case 4:case 5:case 6:case 7:
   case 8:case 9:
      printf ("digit");
      break;
   default:
      printf ("other");
}
```

Case statements rules

- **All values covered by the type of the expression should be covered**

```
   V : Integer;
begin
   case V is
      when 0 =>
         Put_Line (0);
   end case; -- NOK!
```

- **Values must be unique**

```
   V : Integer;
begin
   case V is
      when 0 =>
         Put_Line ("0");
      when Integer'First .. 0 => -- NOK!
         Put_Line ("Negative");
      when others =>
         null;
   end case;
```

Writing ranges for case statements

- **A case statement must contain static ranges only**
  - e.g. ranges computed out of static expressions

```
   V : Integer;
   W : constant Integer := 0;

   subtype I1 is Integer range 1 .. 10;
   subtype I2 is Integer with Static_Predicate => I2 >= 1000;
   subtype I3 is Integer with Dynamic_Predicate => I3 >= V;

   X : Integer;
begin
   case X is
      when V  =>                     -- NOK
      when W  =>                     -- OK
      when I1 =>                     -- OK
      when I2 =>                     -- OK
      when 20 | 30 | 40 =>           -- OK
      when 50 + W =>                 -- OK
      when I3 =>                     -- NOK
      when W + 1 .. Integer'Last =>  -- OK
```

Loop statement

- **Simple loop**

`loop`
`   <statements>`
`   {exit [when <condition>];}`
`   <statements>`
`end loop;`
	No direct equivalent

- **While loop**

`while <condition> loop`
`   <statements>`
`   {exit [when <condition>];}`
`end loop;`
	`while (<condition>)`
	`   <statements>`

- **No do-while/repeat-until loops, use simple loop with exit instead**

For-Loop statement

- **Iteration over indices**
  - range has to be growing
  - var is constant in the loop

`for <var> in <iterator>`
`   |[reverse] <range> loop`
`   <statements>`
`   {exit [when <condition>];}`
`end loop;`
	No direct equivalent

Evaluation of loop range

- **Loop range is evaluated before the loop**

```
   A : Integer := 1;
begin
 for J in A .. F (A) loop
      A := 5; -- We still iterate between 1
              -- and what F(1) returned
 end loop;
```

- **Iterator is constant (can't be modified directly)**

```
for J in 1 .. 10 loop
   J := 5; -- NOK
end loop;
```

```
for (int j = 1; j<=10; j++)
  j = 5;
```

Block statement

- **The *Block* Statement introduces a nested declarative part *and* sequence of statements:**

```
[ declare
      declarative_part ]
begin
      handled_sequence_of_statements
end ;
```

- **The declarative part is optional.**
- **Main uses:**
  – Introduction of local subtypes and arrays that depend on previously computed dynamic values.
  – Local exception handling.

Quiz

Is there an error? (1/10)

```
if A == 0 then
   Put_Line ("A is 0");
end if;
```

Is there an error? (2/10)

```
if A := 0 then
   Put_Line ("A has been assigned to 0");
end if;
```

Is there an error? (3/10)

```
   A : Integer := Integer'Value (Get_Line);
begin
   case A is
      when 1 .. 9 =>
         Put_Line ("Simple digit");
      when 10 .. Integer'Last =>
         Put_Line ("Long positive");
      when Integer'First .. -1 =>
         Put_Line ("Negative");
   end case;
```

Is there an error? (4/10)

```
   A : Integer := Integer'Value (Get_Line);
begin
   case A is
      when Positive =>
         Put_Line ("Positive");
      when Natural =>
         Put_Line ("Natural");
      when others =>
         Put_Line ("Other");
   end case;
```

Is there an error? (5/10)

```
   A : Float := 10.0;
begin
   case A is
      when 1.0 .. Float'Last =>
         Put_Line ("Positive");
      when Float'First .. -1.0 =>
         Put_Line ("Negative");
      when others =>
         Put_Line ("Other");
   end case;
```

Is there an error? (6/10)

```
for I in 0 .. 10 loop
   I := 10;
end loop;
```

What is the output of this code? (7/10)

```
for I in 10 .. 0 loop
   Put_Line (Integer'Image (I));
end loop;
```

Is there an error? (8/10)

```
if A != 0 then
   Put_Line ("A is not 0");
end if;
```

Is there an error? (9/10)

```
   I : Natural;
begin
   for I in 0 .. 10 loop
      null;
   end loop;
```

What is the output of this code? (10/10)

```
   X : Integer := 1;
begin
  for I in 1 .. X loop
      X := 10;
      Put_Line ('A');
  end loop;
```

AdaCore

Arrays

Barnes chapter 8

Arrays are first class citizens

- **All arrays are (doubly) typed**

```ada
type T is array (Integer range <>)
 of Integer;

A : T (0 .. 14);
```

```cpp
int * A = new int [15];
```

- **Properties of array types are…**
  - The index type (can be any discrete type, with optional specific boundaries)
  - The component type (can be any definite type)

- **Properties of array objects are…**
  - The array type
  - Specific boundaries
  - Specific values

Definite vs. Indefinite Types

- **Definite types are types that can be used to create objects without additional information**
  - Their size is known
  - Their constraints are known
- **Indefinite types need additional constraint**
- **Array types can be definite or indefinite**

```
type Definite is array (Integer range 1 .. 10) of Integer;
type Indefinite is array (Integer range <>) of Integer;

A1 : Definite;
A2 : Indefinite (1 .. 20);
```

- **Components of array types must be definite**

Array Indices

- **Array indices can be of any discrete type**
  - Integer (signed or modular)
  - Enumeration
- **Array indices can be defined on any continuous range**
- **Array index range may be empty**

```
type A1 is array (Integer range <>) of Integer;
type A2 is array (Character range 'a' .. 'z') of Integer;
type A3 is array (Integer range 1 .. 0) of Integer;
type A4 is array (Boolean) of Integer;
```

- **Array indices are computed at the point of array type declaration**

```
X : Integer := 0;
type A is array (Integer range 1 .. X) of Integer;
--  changes to X don't change A instances after this point
```

Accessing Array Components

- Array components can be directly accessed

```
   type A is array (Integer range <>) of Integer;
   V : A (1 .. 10);
begin
   V (1) := 0;
```

- Array types and array objects offer ‘Length, ‘Range, ‘First and ‘Last attributes
- On access, bounds are dynamically checked and raise Constraint_Error if overflowed or underflowed

```
   type A is array (Integer range <>);
   V : A (1 .. 10);
begin
   V (0) := 0; -- NOK
```

Array Copy

- **Array operations are first class citizens**

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

   A1 : T (1 .. 10);
   A2 : T (1 .. 10);
begin
   A1 := A2;
```

- **In copy operations, lengths are checked, but not actual indices**

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

   A1 : T (1 .. 10);
   A2 : T (11 .. 20);
   A3 : T (1 .. 20);
begin
   A1 := A2; -- OK
   A1 := A3; -- NOK
```

Array Initialization

- **Array copy can occur at initialization time**

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

A1 : T (1 .. 10);
A2 : T (11 .. 20) := A1;
```

- **If the array type is of an indefinite type, then an object of this type can deduce bounds from initialization**

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

A1 : T (1 .. 10);
A2 : T := A1; -- A2 bounds are 1 .. 10
```

Array Slices

- **It’s possible to refer to only a part of the array using a slice**
  - For array with only one dimension

- **Slices can be used in any place that requires an array object**

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

   A1 : T (1 .. 10);
   A2 : T (1 .. 20);
begin
   A1 := A2 (1 .. 10);
   A1 (2 .. 4) := A2 (5 .. 7);
```

Array Literals (Aggregates)

- **Aggregates can be used to provide values to an array as a whole**

```
({[<position> => ] <expression>,} [others => <expression>])

(1, 2, 3)                           -- finite positional aggregate
(1 => 1, 2 => 10, 3 => 30)  -- finite named aggregate
(1, others => 0)                -- indefinite positional aggregate
(1 => 1, others => 0)          -- indefinite named aggregate
```

- **They can be used wherever an array value is expected**
- **Finite aggregate can initialize variable constraints, lower bound will be equal to T'First**

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

   V1 : T := (1, 2, 3);
   V2 : T := (others => 0); -- NOK (initialization)
begin
   V1 := (others => 0); -- OK (assignment)
```

Array Concatenation

- **Two arrays can be concatenated through the & operators**
  - The resulting array's lower bound is the lower bound of the left operand

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

A1 : T := (1, 2, 3);
A2 : T := (4, 5, 6);
A3 : T := A1 & A2;
```

- **An array can be concatenated with a value**

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

A1 : T := (1, 2, 3);
A2 : T := A1 & 4 & 5;
```

Array Equality

- **Two arrays are equal if**
  - Their Length is equal
  - Their components are equal one by one

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

   A1 : T (1 .. 10);
   A2 : T (1 .. 20);

begin

   if A1 = A2 then -- ALWAYS FALSE
```

- **Actual indices do not matter in array equality**

Arrays are first class citizens (2)

- **All array types can be passed as formal parameters to/from subprograms.**

- **Array types can be returned from a function.**
  - Function return is *by-copy*, so can impose some performance penalty.
    - Alternative: use a procedure with an out parameter – almost certainly passed *by-reference*, so efficient.

- **A function can even return an unconstrained array type, like String.**

Loops over an array

- **Through an index loop**

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

A : T (1 .. 10);

for I in A'Range loop
   A (I) := 0;
end loop;
```

Matrices

- **Two dimensional arrays**

```
   type T is array (Integer range <>, Integer range <>) of Integer;
   V : T (1 .. 10, 0 .. 2);
begin
   V (1, 0) := 0;
```

  – Attributes are ’First (dimension), ’Last (dimension), ’Range (dimension)

- **Arrays of arrays**

```
   type T1 is array (Integer range <>) of Integer;
   type T2 is array (Integer range <>) of T1 (0 .. 2);
   V : T (1 .. 10);
begin
   V (1)(0) := 0;
```

Strings

- **Strings are regular arrays. Type String is declared in package Standard**

```
type String is array (Positive range <>) of Character;
```

- **There is a special String literal**

```
V  : String := "This is it";
V2 : String := "Here come quotes ("")";
```

- **The package ASCII provides named Character constants.**

```
V  : String := "This is null terminated" & ASCII.NUL;
```

- **In Ada95 onwards, you can also use Ada.Characters.Latin_1 and siblings.**

Array Subtypes and Derived Types

- When subtyping an array, it's possible to define a constraint

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

subtype One_To_Ten is Any_Bounds (1 .. 10);
```

- Same with array derivation

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

type One_To_Ten is new Any_Bounds (1 .. 10);
```

- Once the array is definite, bounds cannot be changed

Quiz

Is there an error? (1/10)

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

    type T is array (My_Int) of Integer;

    V : T;
begin
    V (1) := 2;
```

Is there an error? (2/10)

```
   type T is array (Integer) of Integer;

   V : T;
begin
   V (1) := 2;
```

Is there an error? (3/10)

```
   type T1 is array (Integer range <>) of Integer;
   type T2 is array (Integer range <>) of Integer;

   V1 : T1 (1 .. 3) := (others => 0);
   V2 : T2 := (1, 2, 3);
begin
   V1 := V2;
```

Is there an error? (4/10)

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

   V : T := (1, 2, 3);
begin
   V (0) := V (1) + V (2);
```

Is there an error? (5/10)

```
   type T is array (Integer range <>) of Integer;
   subtype TS is T (1 .. 2);

   V1 : T (10 .. 11);
   V2 : TS := (others => 0);
begin
   V1 := V2;
```

Is there an error? (6/10)

```
   X : Integer := 10;
   type T is array (Integer range 1 .. X) of Integer;
   V1 : T;
begin
   X := 100;
   declare
      V2 : T;
   begin
      V1 := V2;
```

Is there an error? (7/10)

```
   type T is array (Integer range <>) of Integer;
   V1 : T (1 .. 3) := (10, 20, 30);
   V2 : T := (10, 20, 30):
begin
   for I in V1'Range loop
      V1 (I) := V1 (I) + V2 (I);
   end loop;
```

Is there an error? (8/10)

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

subtype TS is Any_Bounds (1 .. 10);

type T2 is new TS (1 .. 9);
```

Is there an error? (9/10)

```
type String_Array is array (Integer range <>) of String;
```

Is there an error? (10/10)

```
X : Integer := 0;

type T is array (Integer range <>) of Integer
   with Default_Component_Value => X;

V : T (1 .. 10);
```

Record types

Barnes chapter 8

Record types

- **Allow named heterogeneous data in a type**

```
type Shape is record
   Id   : Integer;
   X, Y : Float;
end record;
```

- **Fields are accessed through dot notation**

```
   S : Shape;
begin
   S.X := 0.0;
   S.Id := 1;
```

Record types

- **Any definite type can be used as a component type**

```
type Position is record
   X, Y : Integer;
end record;

type Shape is record
   Name : String (1 .. 10);
   P    : Position;
end record;
```

- **Size may not be known at compile time**

```
Len : Natural := Compute_Len;
type Name_Type is String (1 .. Len);

type Shape is record
   Name : Name_Type;
   P    : Position;
end record;
```

  – Has impact on code generated

Default Values

- **Default values can be provided to record components:**

```
type Position is record
   X : Integer := 0;
   Y : Integer := 0;
end record;
```

- **Default values are dynamic expressions evaluated at each object declaration**

```
Cx, Cy : Integer := 0;

type Position is record
   X : Integer := Cx;
   Y : Integer := Cy;
end record;

P1 : Position; -- = (0, 0);

begin

   Cx := 1;
   Cy := 1;

   declare

      P2 : Position; -- = (1, 1);
```

Aggregates (1/2)

- **Like arrays, record values can be given through aggregates**

```ada
type Position is record
   X, Y : Integer;
end record;

type Shape is record
   Name : String (1 .. 10);
   P    : Position;
end record;

Center : Position := (0, 0);
Circle : Shape := ((others => ' '), Center);
```

- **Named aggregates are possible (but cannot switch back to positional)**

```ada
P1 : Position := (0, Y => 0);        -- OK
P1 : Position := (X => 0, Y => 0); -- OK
P3 : Position := (Y => 0, X => 0); -- OK
P4 : Position := (X => 0, 0);        -- NOK
```

Aggregates (2/2)

- **Named aggregate is required for one-element records**

```
type Singleton is record
   V : Integer;
end record;

V1 : Singleton := (V => 0); -- OK
V2 : Singleton := (0);      -- NOK
```

- **Default values can be referred as <> after a name or *others***

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

V1 : Rec := (others => <>); -- QUIZ is this OK?
V2 : Rec := (A => 0, B => <>, others => <>);
```

- **If all remaining types are the same, *others* can use an expression**

```
type Rec is record
   A, B : Integer;
   C, D : Float;
end record;

V1 : Rec := (0, 0, others => 0.0);
```

Discriminant problematic

- **Only a subset of the components are needed to use this type, depending on the context**

```
type Shape is record
   X, Y         : Float;
   X2, Y2       : Float;
   Radius       : Float;
   Outer_Radius : Float;
end record;
```

- **Why do we need to use the memory for Radius if the shape is a line?**

Use of a discriminant

- **Types can be parameterized by a discrete type**

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

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

- **This type is *indefinite*, so needs to be constrained at object declaration**

```
V : Shape (Circle);
```

General Syntax

```
type Id ([Discriminant : Discrete_Type] {, Discriminant : Discrete_Type}) is
   record
   [common part]

   [variant part]
end record;
```

- **All identifiers must be unique – even if declared in distinct variant parts**
- **There can be a variant part within the variant part**
- **All values must have a branch in the case – use *others* if needed**
- **The object will fit the size needed to work with the given discriminant – unnecessary fields won't get allocated**

Usage of a record with discriminant

- **As for arrays – the unconstrained part has to be specified**

```
   V1 : Shape (Circle);
   V2 : Shape := V1; -- OK, constrained by initialization
begin
   V1.Radius := 0.0; --  OK, radius is in the Circle case
   V2.X2 := 0.0;     --  Raises constraint error
```

- **Accessing a component not accessible for a given constraint will raise *Constraint_Error***

Aggregates

- **Same as record aggregates – but have to give a value to the discriminant**

- **Only the values related to the constraint have to be valuated**

```
V1 : Shape := (Kind => Line,
               X    => 0.0,
               Y    => 0.0,
               X2   => 10.0,
               Y2   => 10.0);

V2 : Shape := (Circle, 0.0, 0.0, 5.0);
```

Constraints on record components

- **Record component types need to be definite**

- **If a constraint is needed, it can be dependent on the discriminant value**

```
type String_Container (Size : Positive) is record
   S : String (1 .. Size);
end record;

V : String_Container (20);
```

Mutable objects (1/2)

- We may want to change the constraint of an object over time
- Such objects need to have an default initial value for their discriminants – they are *constrained*
- The discriminant can't be changed on its own – the whole object has to be assigned to a new value
- The discriminant of an object with an explicit constraint can't be changed

```
    type Shape (Kind : Shape_Kind := Line) is record
     ...
    end record;

    V  : Shape (Circle); -- Still Ok
    V2 : Shape;          --  Ok, of type line
begin
    V2 := V;                  --  OK, since the object is mutable
    V  := (Line, 0.0, 0.0, 0.0, 0.0);
      -- Raises Constraint_Error, V has been explicitly constrained
```

Mutable objects (2/2)

- The size of a mutable object is the *maximal* size needed to represent all possible objects
- Be careful when used with array constraints !

```
type String_Container (Size : Positive := 1) is record
   S : String (1 .. Size);
end record;

V : String_Container;
```

- The above might raise Storage_Error, since the maximal size is enough memory to store Positive'Last characters.

Quiz

Is there an error? (1/10)

```
type R is record
   A, B, C : Integer := 0;
end record;

V : R := (A => 1);
```

Is there an error? (2/10)

```
type My_Integer is new Integer;

type R is record
   A, B, C : Integer := 0;
   D       : My_Integer := 0;
end record;

V : R := (others => 1);
```

Is there an error? (3/10)

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

Is there an error? (4/10)

```
type My_Integer is new Integer;

type R is record
   A, B, C : Integer;
   D       : My_Integer;
end record;

V : R := (others => <>);
```

Is there an error? (5/10)

```
type R is record
   A : Integer := 0;
end record;

V : R := (0);
```

Is there an error? (6/10)

```
type R is record
   V : String;
end record;

V : R := (V => "Hello");
```

Is there an error? (7/10)

```
    type R (D : Integer) is record
       null;
    end record;

    V1 : R := (D => 5);
    V2 : R := (D => 6);
begin
    V1 := V2;
```

Is there an error? (8/10)

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

V : R := (5, "Hello");
```

Is there an error? (9/10)

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

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

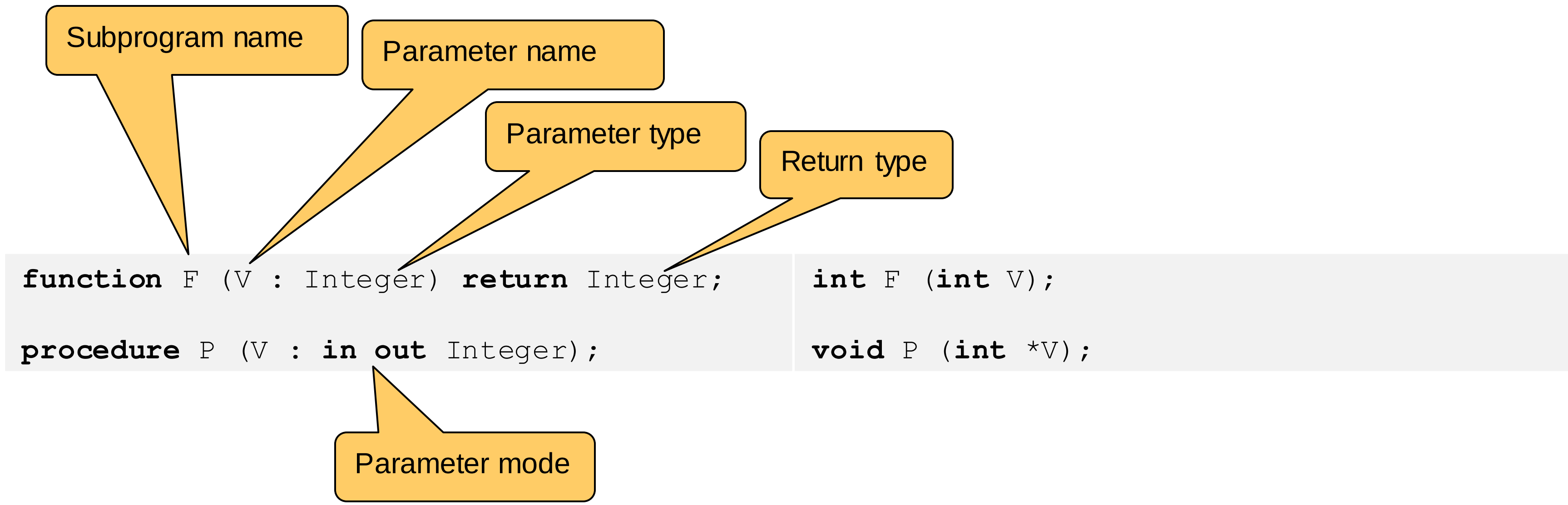
Is there an error? (10/10)

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

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

    V  : Shape := (Circle, others => <>);
    V2 : Shape := (Line, others => <>);
begin
    V := V2;
```

Subprograms

Barnes chapter 10

Subprograms in Ada: Specifications




```ada
function F (V : Integer) return Integer;

procedure P (V : in out Integer);
```

```c
int F (int V);

void P (int *V);
```

- **Ada differentiates functions (returning values) and procedures (with no return values)**
  - A function call is an *expression*.
  - A procedure call is a *statement.*

Subprograms in Ada: Declaration and Body

Declaration

```
function F (V : Integer) return Integer;
```

Body

```
function F (V : Integer) return Integer is
   R : Integer := V * 2;
begin
   R := R * 2;
   return R - 1;
end F;
```

- **Declaration is optional, but must be given before use**
- **Functions' result cannot be ignored**
- **Completion / body is introduced by "is"**

Parameter Modes

- **Mode "in"**
  - Actual parameter is not altered
  - Only reading of formals is allowed
  - Default mode
- **Mode "out"**
  - Actual is expected to be altered
  - Writing is expected, but reading is also allowed
  - Initial value is not defined
  - Only for **procedure**
- **Mode "in out"**
  - Actual is expected to be both read and altered
  - Both reading & updating of formals is allowed
  - Only for **procedure**

```
function F (V : in Integer) return Integer is
   R : Integer := V * 2;
begin
   return R - 1;
end F;

procedure P (V : in out Integer) is
begin
   V := 0;
end P;
```

Parameter Passing Mechanisms

- **Passed either “by-copy” or “by-reference”**
- **By-Copy**
  - The formal denotes a separate object from the actual
  - A copy of the actual is placed into the formal before the call
  - A copy of the formal is placed back into the actual after the call
- **By-Reference**
  - The formal denotes a view of the actual
  - Reads and updates to the formal directly affect the actual
- **Parameter types control mechanism selection**
  - Not the parameter modes

Standardized Parameter Passing Rules

- **By-Copy types**
  - Scalar types
  - Access types
  - Private types that are fully defined as by-copy types
- **By-Reference types**
  - Tagged types
  - Task types and Protected types
  - Limited types
  - Composite types with by-reference component types
  - Private types that are fully defined as by-reference types
- **Implementation-defined types**
  - Array types containing only by-copy components
  - Non-limited record types containing only by-copy components
  - Implementation chooses most efficient method

Subprogram Calls

- If no parameter is given, no parenthesis is allowed

```
function F return Integer;

V : Integer := F;
```

- Named parameter association is possible

```
procedure P (A, B, C : Integer);

P (B => 0, C => 0, A => 1);
```

- *out* and *in out* modes require a *variable* object

```
procedure P (X : out Integer);

V  : Integer;
VC : constant Integer := 1;

P (V);  -- OK
P (VC); -- NOT OK
```

Default Values

- "in" parameters can be provided with a default value

```
procedure P (A : Integer := 0; B : Integer := 0);
```

- Default values are dynamic expressions, evaluated at the point of call if no explicit expression is given

```
P;             -- A = 0, B = 0;
P (1);         -- A = 1, B = 0;
P (B => 2);    -- A = 0, B = 2;
P (1, 2);      -- A = 1, B = 2;
```

Indefinite Parameters and Return Types

- Subprograms can have indefinite parameters and return types

```
function Comment (Stmt : String) return String is
begin
   return "/*" & Stmt & "*/";
end Comment;

S : String := Comment ("a=0"); -- return /*a=0*/
```

- Constraints are computed at the point of call
- Don’t assume boundaries!

```
   procedure Init (Stmt : in out String) is
   begin
      for J in 1 .. Stmt'Length loop
         Stmt (J) := ' ';
      end loop;
   end Init;

   S : String := "ABCxxx";
begin
   Init (S (4 .. 6));
```

Overloading (1/2)

- **Ada allows overloading of subprograms**

```
procedure Print (V : Integer);
procedure Print (V : Float);
```

- **Overloading is allowed if specifications differ by**
  - Number of parameters
  - Type of parameters
  - Result type

```
subtype Positive is Integer range 1 .. Integer'Last;
procedure Print (V : Integer);
procedure Print (W : out Positive); -- NOK
```

- **Some aspects of the specification are not taken into account**
  - Parameter names
  - Parameter subtypes
  - Parameter modes
  - Parameter default expressions

Overloading (2/2)

- Overloading may introduce ambiguities at call time
- Ambiguities can be solved with additional information

```
    type Apples is new Integer;
    type Oranges is new Integer;


    procedure Print (Nb_Apples : Apples);
    procedure Print (Nb_Oranges : Oranges);

    N_A : Apples := 0;

begin

    Print (N_A);                -- OK
    Print (0);                  -- NOK
    Print (Oranges'(0));        -- OK
    Print (Nb_Oranges => 0);    -- OK
```

Operator Overloading

- **Default operators (=, /=, *, /, +, -, >, <, >=, <=, and, or…) can be overloaded, added or removed for types**

```
type Distance is new Float;
type Surface is new Float;

function "*" (L, R : Distance) return Distance is abstract; -- removes "*"
function "*" (L, R : Surface) return Surface is abstract;   -- removes "*"

-- Add "*" for (Distance, Distance) -> Surface
function "*" (L, R : Distance) return Surface;

type R is record
   Unimportant_Field : Integer;
   Important_Field   : Integer;
end record;

function "=" (Left, Right : R) return Boolean is
begin
   return Left.Important_Field = Right.Important_Field;
end "=";
```

- **"=" overloading will automatically generate the corresponding "/="**

Hiding

- **It is possible to declare two subprograms of the exact same profile but in different scope**
- **Overloading rules don’t apply here - the nested subprogram hides the one declared in the parent scope**

```
A : declare
   procedure P (V : Integer);
begin
   P (0); -- calls A.P

   B : declare
      procedure P (V : Integer);
   begin
      P (0); -- calls B.P
      A.P (0); -- calls A.P
```

- **This is considered bad practice**

Nested Subprograms and Access to Globals

- **A subprogram can be nested in any scope**
- **A nested subprogram will have access to the parent subprogram parameters, and variables declared before**

```
procedure P (V : Integer) is
   W : Integer;

   procedure Nested is
   begin
      W := V + 1;
   end Nested;
begin
   W := 0;
   Nested;
```

Quiz

Is there an error? (1/10)

```
   function F (V : Integer) return Integer is
   begin
      Put_Line (Integer'Image (V));
      return V + 1;
   end F;

begin

   F (999);
```

Is there an error? (2/10)

```
procedure P (V : Integer) is
begin
   V := V + 1;
end P;
```

Is there an error? (3/10)

```
function F () return Integer is
   return 0;
end F;

V : Integer := F ();
```

Is there an error? (4/10)

```
procedure P (V : Integer) is
   procedure Nested is
   begin
      W := V + 1;
   end Nested;

   W : Integer;
begin
   W := 0;
   Nested;
```

Is there an error? (5/10)

```
function F return String is
begin
   return "A STRING";
end F;

V : String (1 .. 2) := F;
```

Is there an error? (6/10)

```
   procedure P (V : Integer := 0);
   procedure P (V : Float := 0.0);
begin
   P;
```

Is there an error? (7/10)

```
   procedure P1 (V : Integer := 0) is … end;

   procedure P2 (V : Integer := 0) is … end;
begin
   declare
      procedure P1 (V : Integer := 0) is … end;

      procedure P2 (V : Float := 0.0) is … end;
   begin
      P1;
      P2;
   end;
```

Is there an error? (8/10)

```
   procedure Multiply (V : out Integer; Times : Integer) is
   begin
      for J in 1 .. Times loop
         V := V + V;
      end loop;
   end Multiply;

   X : Integer := 10;
begin
   Multiply (X, 50);
```

Is there an error? (9/10)

```
   type My_Int is new Integer;

   function "=" (L, R : My_Int) return Boolean;

   function "=" (L, R : My_Int) return Boolean is
   begin
      if L <= 0 or else R <= 0 then
         return True;
      else
         return L = R;
      end if;
   end "=";

   V, W : My_Int := 1;
begin
   if V = W then
...
```

Is there an error? (10/10)

```
    type My_Int is new Integer;

    function "=" (L, R : My_Int) return Boolean;

    function "=" (L, R : My_Int) return Boolean is ...

    A, B : My_Int;
begin
    if A /= B then
    ...
```

Packages

Barnes chapters 12, 13

The Ada Package

- A package is the base of software architecture in Ada
- It's a semantic entity checked by the compiler
- It separates clearly a specification and an implementation

```ada
-- p.ads

package P is
   procedure Proc;
end P;

-- p.adb

package body P is
   procedure Proc is
   begin
      null;
   end Proc;
end P;
```

```c
/* p.h */

#ifndef __P_H__
#define __P_H__

void Proc ();

#endif

/* p.c */

int V;
void Proc () {
}
```

General Structure of a Package

```
package P is
   --  Public part of the specification.
   --  Declaration of subprograms, variables exceptions, tasks.
   --  Visible to the external user.
   --  Used by the compiler for all dependencies.
end P;

package body P is
  --  Body
  --  Declaration of subprograms, variables exceptions, tasks.
  --  Implementation of subprograms.
  --  Used by the compiler to generate code for P.
  --  In certain cases (e.g. Inlining and Generics), used by the
  --  compiler to compile clients of P.
end P;
```

- Entities should be put in the body except if they have to be exported
- The body is easier to change than the specification

Uses of a Package

1. **Provide a common naming space for a logically related set of entities**
   - The package acts as a name wrapper
   - These kind of packages are typically stateless (i.e. there are no global objects)
2. **Group related types and objects**
   - A package of this sort provides a single place for inter-related types and objects
   - This type of package does not typically have a body
3. **One-of-a-kind (aka "singleton") objects**
   - One-of-a-kind objects are objects for which a single instance exists
   - One-of-a-kind packages have the object state in their body
4. **Create a data type abstraction**
   - Also known as "Abstract Data Type" (ADT)
   - An ADT is a data type T (or family thereof) together with the operations that are allowed to manipulate objects of type T

Accessing components of a package

- **Only entities declared in the public part are visible**
- **Entities are referenced through the dot notation**

```
package P1 is

   procedure Pub_Proc;

end P1;
```

```
package body P1 is

  procedure Priv_Proc;
  …
end P1;
```

```
package P2 is

   procedure Proc;

end P2;
```

```
with P1;

package body P2 is

   procedure Proc is
   begin
      P1.Pub_Proc;
      P1.Priv_Proc;
   end Proc;

end P2;
```

Child units

- A public child unit is an extension of a package
- Can be used to organize the namespace or break big packages into pieces
- Child units have visibility over parents

```
-- p.ads
package P is

end P;
```

```
-- p-child_1.ads
package P.Child_1 is

end P.Child_1;
```

```
-- p-child_2.ads
package P.Child_2 is

end P.Child_2;
```

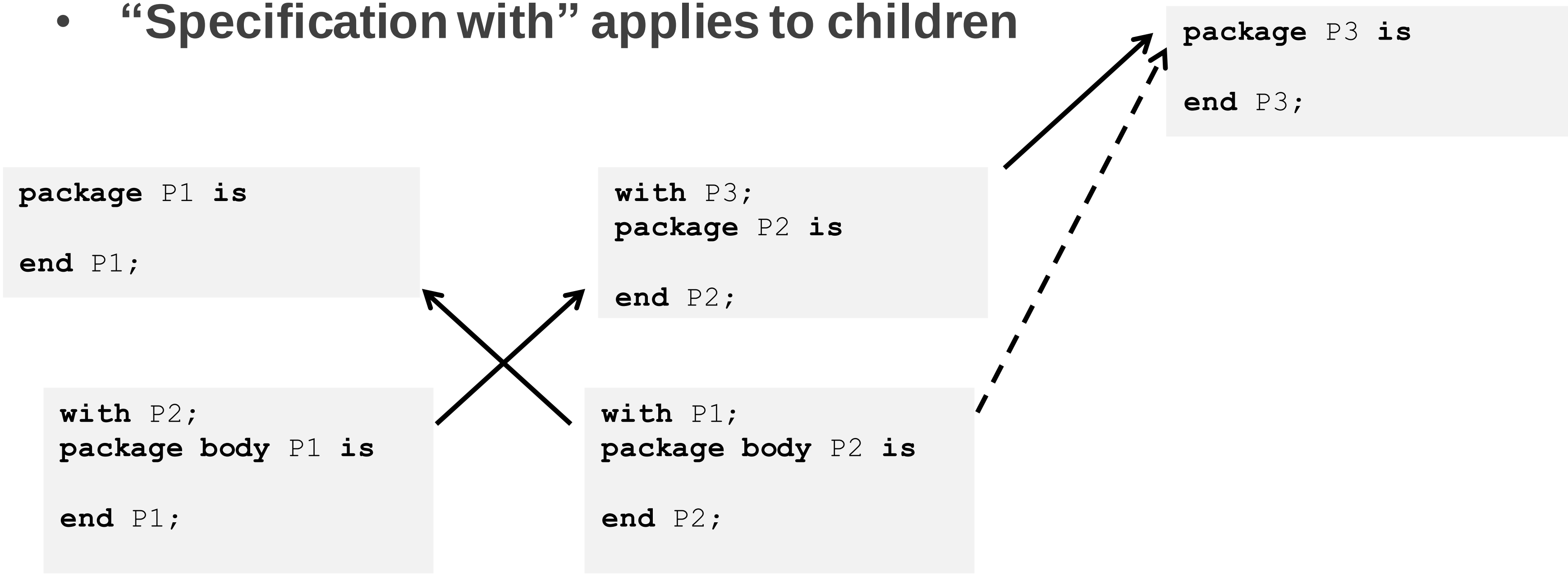
```
-- p-child_3.ads
package P.Child_3 is

end P.Child_3;
```

```
-- p-child_2-grand_child.ads
package P.Child_2.Grand_Child is

end P.Child_2.Grand_Child;
```

- Generally speaking, it's a good habit to split functionality into packages as much as possible

Full dependencies ("with clause")

- "With clause" defines a dependency between two packages
- Gives access to all the public declarations
- Can be applied to the spec or the body
- A dependency is normally done to a specification
- "Specification with" applies to the body
- "Specification with" applies to children



Partial dependencies ("limited with")

- Circular dependencies between units are forbidden (to avoid illegal circular constructions)

```
with Medical;
package Person is
   type Person_R is record
      T_Info : Medical.Medical_R;
   end record;
end Person;
```

```
with Person;
package Medical is
   type Medical_R is record
      T_Info : Person.Person_R;
   end record;
end Person;
```

- A partial dependency ("limited with") allows such circularity, but gives visibility of an incomplete view of type declarations only (see later for more details)

```
limited with Medical;
package Person is
   type Person_R is record
      T_Info : access Medical.Medical_R;
   end record;
end Person;
```

```
limited with Person;
package Medical is
   type Medical_R is record
      T_Info : access Person.Person_R;
   end record;
end Person;
```

- Regular "with clauses" can still be used in bodies

Dependency shortcut (“use clause”)

- Prefix may be overkill
- The “use clause” grants “direct visibility” so the prefix can be omitted.
- Can introduce ambiguities
- Can be placed in any scope

```
package P1 is

   procedure Proc1;
   type T is null record;

end P1;
```

```
package P2 is

   procedure Proc1;

end P2;
```

```
with P1;
with P2; use P2;

package body P3 is

   X : T;          -- ✗

   procedure Proc is
      use P1;
      X : T;
   begin
      Proc1;       -- ✗
      P1.Proc1;
      P2.Proc1;
   end Proc;

end P3;
```

A Package is a High Level Semantic Entity

- **The compiler is responsible for checking structural and semantic consistency**

```ada
-- p.ads

package P is
   V : Integer;
   procedure Proc;
   pragma Inline (Proc);
end P;

-- p.adb

package body P is
   procedure Proc is
   begin
      null;
   end Proc;
end P;
```

```c
/* p.h */

#ifndef __P_H__
#define __P_H__

extern int V;
inline void Proc ();

#include "p.hi"
#endif

/* p.hi */

#ifndef __P_HI__
#define __P_HI__

inline void Proc () {
}

#endif

/* p.c */

int V;
```

Compilation with GNAT (1/2)

- **The compiler knows how to work just with the specification**

```ada
package Dep1 is

end Dep1;
```

```ada
package P is

end P;
```

```ada
package Dep2 is

end Dep2;
```

```ada
package body Dep1 is

end Dep1;
```

```ada
with Dep1;
with Dep2;

package body P is

end P;
```

```ada
Package body Dep2 is

end Dep2;
```

```
gcc -c p.adb
```

Compilation with GNAT (2/2)

- **If information is needed from the body (generic, inline), the compiler works transparently**

```
package Dep1 is
   procedure Proc;
   pragma Inline (Proc);
end Dep1;
```

```
package P is

end P;
```

```
package Dep2 is

end Dep2;
```

```
package body Dep1 is

end Dep1;
```

```
with Dep1;
with Dep2;

package body P is

end P;
```

```
package body Dep2 is

end Dep2;
```

```
gcc -c p.adb
```

Quiz

Is there a compilation error? (1/10)

```
package P1 is
   type T is null record;
end P1;
```

```
package P2 is
   X : P1.T;
end P2;
```

Is there a compilation error? (2/10)

```
package P1 is

end P1;
```

```
with P1; use P1;

package P2 is

   X : T;

end P2;
```

```
package body P1 is

   type T is null record;

end P1;
```

Is there a compilation error? (3/10)

```
with P2;

package P1 is

   type T1 is null record;

   V : P2.T2;

end P1;
```

```
with P1;

package P2 is

   type T2 is null record;

   V : P1.T1;

end P2;
```

Is there a compilation error? (4/10)

```
with P2;

package P1 is

   type T1 is null record;

   V : P2.T2;

end P1;
```

```
limited with P1;

package P2 is

   type T2 is null record;

   V : access P1.T1;

end P2;
```

Is there a compilation error? (5/10)

```
with P2;

package P1 is

   type T1 is null record;

   V : P2.T2;

end P1;
```

```
package P2 is

   type T2 is null record;

end P2;
```

```
with P1;

package body P2 is

   X : P1.T1;

end P2;
```

Is there a compilation error? (6/10)

```
package P1 is
   type T is null record;
end P1;
```

```
package P1.Child is
end P1.Child;
```

```
package body P1.Child is
   X : T;
end P1.Child;
```

Is there a compilation error? (7/10)

```
with P1.Child;

package P1 is

   X : P1.Child.T;

end P1;
```

```
package P1.Child is

   type T is null record;

end P1.Child;
```

Is there a compilation error? (8/10)

```
package P1 is

end P1;
```

```
package P1.Child is

   type T is null record;

end P1.Child;
```

```
with P1.Child;

package body P1 is

   X : P1.Child.T;

end P1;
```

Is there a compilation error? (9/10)

```ada
limited with P2;

package P1 is

   type T1 is null record;

   V : P2.T2;

end P1;
```

```ada
limited with P1;

package P2 is

   type T2 is null record;

   V : access P1.T1;

end P2;
```

Is there a compilation error? (10/10)

```
package Dep is

   type T is null record;

end Dep;
```

```
with Dep;

package P1 is


end P1;
```

```
package P1.Child is

end P1.Child;
```

```
package body P1.Child is

   X : Dep.T;

end P1.Child;
```

Basic Privacy

Barnes chapter 12

Private types

Typical problem

- **Having the full implementation of the types accessible is error-prone**

```
package Stacks is

   type Stack_Data is array (1 .. 100) of Integer;

   type Stack_Type is record
      Max  : Integer := 0;
      Data : Stack_Data;
   end record;

   procedure Push
      (Stack : in out Stack_Type; Val : Integer);

   procedure Pop
      (Stack : in out Stack_Type; Val : out Integer);

end Stacks;
```

```
procedure Main is
   S : Stacks.Stack_Type;
   V : Integer;
begin
   Push (S, 15);
   S.Max := 10;
   Pop (S, V);
end Main;
```

- **But the compiler needs to have access to the representation (needs to know how much memory is to be used)**
- **So the representation has to stay in the specification**

Private types

- **Introduces a new section in the package specification : the private section**
  - Visible to the compiler
  - Visible to the body and any child packages
  - **Not visible to the user of the package**
- **In Ada, private applies to a type as a whole, not on a field by field basis**
- **In Ada, privacy is managed at package level, not at class level**

```ada
package Stacks is

   type Stack_Type is private;

   procedure Push
      (Stack : in out Stack_Type;
       Val   : Integer);

private

  type Stack_Data is array (1 .. 100)
      of Integer;

   type Stack_Type is record
      Max  : Integer := 0;
      Data : Stack_Data;
   end record;

end Stacks;
```

```cpp
namespace Stacks {

   class Stack_Type {
      public:
         void Push (int val);

      private:
         int [] Data;
         int Max;
   };

}
```

Who has access to the private information?

- **Body, and child units have access to the implementation**

```
package Stacks is
   type Stack_Type is private;

   procedure Push
       (Stack : in out Stack_Type;
        Val   : Integer);
private
   type Stack_Data is array (1 .. 100)
      of Integer;

   type Stack_Type is record
      Max  : Integer := 0;
      Data : Stack_Data;
   end record;
end Stacks;

package body Stacks is
  procedure Push
       (Stack : in out Stack_Type;
        Val   : Integer)
  is
  begin
      Stack.Data (Stack.Max + 1) := Val;
      Stack.Max := Stack.Max + 1;
  end Push;
end Stacks;
```

```
package Stacks.Utils is
   procedure Empty
      (Stack : in out Stack_Type);
end Stacks.Utils;

package body Stack.Utils is
   procedure Empty
      (Stack : in out Stack_Type) is
   begin
      Stack.Max := 0;
   end Stack.Utils;
end Stack.Utils;
```

```
with Stacks;       use Stacks;
with Stacks.Utils; use Stacks.Utils;

procedure Main is
   S : Stack_Type;
begin
   Push (S, 10);
   Empty (S);
   S.Max := 0;
end Main;
```

What can you do with a private type?

- **From the user perspective, a private type is equivalent to a null record**
- **It can be used for**
  - Variables, parameters and components declarations
  - Copies (“:=” is predefined)
  - Comparisons (“=“ and “/=“)

```
package Stacks is

   type Stack_Type is private;
   procedure Push
      (Stack : in out Stack_Type;
       Val   : Integer);

private

  […]

end Stacks;
```

```
procedure Main is
   S1, S2 : Stacks.Stack_Type;
begin
   Push (S1, 15);
   S2 := S1;

   Push (S2, 0);
   Push (S1, 0);

   if S1 = S2 then
      Push (S1, 1);
   end if;
end Main;
```

How can a private type be implemented?

- **A "simple" private type can be implemented by any type giving at least the same level of capabilities**
  - The type must allow variable declarations without the need of constraints, it has to be definite (e.g. no unconstrained arrays)
  - The type must allow copy and comparison (e.g. no limited types)

```
package Stacks is

   type Stack_Type is private;
```

```
private

   type Stack_Type is range 1 .. 10;

end Stacks;
```

```
private

   type Stack_Type is record
      V : Integer;
   end record;

end Stacks;
```

```
private

   type Stack_Type is array
       (Integer range 1 .. 10);
        of Integer;

end Stacks;
```

```
private

   type Stack_Type (Size : Integer) is record
      V : Integer;
   end record;
end Stacks;
```

```
private

   type Stack_Type is array (Integer range <>)
         of Integer;

end Stacks;
```

How can a private type be implemented?

- **An “indefinite” private type can be implemented by any type that can be implemented by private type as well as indefinites**
  - But the user needs to consider it as indefinite (no declaration without initialization)

```
package Stacks is

   type Stack_Type (<>) is private;
```

```
private

   type Stack_Type is range 1 .. 10;

end Stacks;
```

```
private

   type Stack_Type is record
      V : Integer;
   end record;

end Stacks;
```

```
private

   type Stack_Type is array
      (Integer range 1 .. 10);
       of Integer;

end Stacks;
```

```
private

   type Stack_Type (Size : Integer) is record
      V : Integer;
   end record;
end Stacks;
```

```
private

   type Stack_Type is array (Integer range <>)
       of Integer;

end Stacks;
```

Public Discriminants on Private Types

- **It's possible to specify the discriminants of a private type**

```
package Stacks is

  type Stack_Type (Size : Integer) is private;

private

   type Stack_Type (Size : Integer) is record
      V : Integer;
   end record;
end Stacks;
```

Deferred private constants

- **It's useful to declare constants visible in the public view**
- **Values can't be given before the representation is accessible – so constants of private types have a public and a private view**

```
package Stacks is
   type Stack_Type is private;

   Empty_Stack : constant Stack_Type;

private

   type Stack_Data is array (1 .. 100)
      of Integer;

   type Stack_Type is record
      Max  : Integer := 0;
      Data : Stack_Data;
   end record;

   Empty_Stack : constant Stack_Type :=
      (0, (others => 0));
end Stacks;
```

Private part is not only for private types

- **Any kind of declaration can be provided in the private part of the package**
- **Entities declared only in the private part are not visible at all to a client**

```
package P is
   --  Public part of the specification.
   --  Declaration of subprograms, variables exceptions, tasks.
   --  Visible to the external user
   --  Used by the compiler for all dependencies.
private
   --  Private part of the specification.
   --  Declaration of subprograms, variables exceptions, tasks.
   --  Visible to the children and the implementation.
   --  Used by the compiler for all dependencies.
end P;

package body P is
  --  Body
  --  Declaration of subprograms, variables exceptions, tasks.
  --  Implementation of subprograms
end P;
```

Quiz

Is there a compilation error? (1/7)

```
package P is
   type T is private;
private
   type T is range 0 .. 10;
end P;
```

```
with P; use P;

procedure P.Main is
   V : T;
begin
   V := 0;
end P.Main;
```

```
with P; use P;

procedure Main is
   V : T;
begin
   V := 0;
end Main;
```

Is there a compilation error? (2/7)

```
package P is
   type T is private;
   Zero : constant T := 0;
private
   type T is range 0 .. 10;
end P;
```

```
with P; use P;

package P2 is
   type T2 is record
      F : T;
   end record;
end P2;
```

```
with P;  use P;
with P2; use P2;

procedure Main is
   V : T2;
begin
   V.F := Zero;
end Main;
```

Is there a compilation error? (3/7)

```
package P is
   type T is private;
private
   type T is range 0 .. 10;
   Zero : constant T := 0;
end P;
```

```
with P; use P;

procedure P.Main is
   V : T;
begin
   V := Zero;
end P.Main;
```

```
with P; use P;

procedure Main is
   V : T;
begin
   V := Zero;
end Main;
```

Is there a compilation error? (4/7)

```
package P is
   type T is private;
private
   type T is array (Integer range <>) of Integer;
end P;
```

```
procedure P.Main is
   V : T (1 .. 10);
begin
   V (1) := 0;
end P.Main;
```

Is there a compilation error? (5/7)

```
package P is
   type T (<>) is private;
private
   type T is array (Integer range 1 .. 10) of Integer;
end P;
```

```
with P; use P;

procedure Main is
   V : T;
begin
   null;
end Main;
```

Is there a compilation error? (6/7)

```
package P is
   type T is private;

   One : constant T;
private
   type T is range 0 .. 10;
   One : constant T := 0;
end P;
```

```
with P; use P;

procedure Main is
   Val : T;
begin
   Val := One + One;
end Main;
```

Is there a compilation error? (7/7)

```
package P is
   type T is private;
private
   type T is range 0 .. 10;
end P;
```

```
package P.Constants is
   Zero : constant T := 0;
   One  : constant T := 1;
end P.Constants;
```

```
with P;           use P;
with P.Constants; use P.Constants;

procedure Main is
   V : T := One;
begin
   null;
end Main;
```

Exceptions

Barnes chapter 15

Exception Declaration and Raise

- **Ada exceptions are a dedicated kind of entity**
  - associated with a scope and visibility
  - declared like a variable

```
My_Exception : exception;
```

- **The environment can raise predefined exceptions**
  - Constraint_Error
  - Program_Error
  - Storage_Error
  - …

Manual Exception Raise

- **An exception can be raised manually, and associated with a message**
  - As a raise statement

```
raise My_Exception;
raise My_Exception with "My message";
```

Exception Handling

- **Exception can be caught at the end of any block of statements**

```
begin
   -- some code
exception
   when My_Exception =>
      -- some code
end;
```

```
try {
   // some code
} catch (My_Exception e) {
   // some code
}
```

- **Several exceptions can be handled by the same code**

```
begin
   -- some code
exception
   when Constraint_Error | Storage_Error =>
      -- some code
   when others =>
      -- code for all other exceptions
end;
```

Exceptions Occurences and Reraise

- In an exception block, the current exception can be re-raised

```
exception
   when others =>
       raise;
end;
```

- It is possible to manipulate the current occurrence by naming it, allowing its message to be extracted or to re-raise an occurrence explicitly

```
with Ada.Exceptions; use Ada.Exceptions;

[...]

exception
   when E : others =>
      Put_Line (Exception_Message (E));
      Reraise_Occurrence (E);
end;
```

Class Exercise

- **In the Ada RM, find and have a look at the specification of the package**
  - Ada.Exceptions

- **In the GNAT Runtime Sources, find and have a look at the specification of the package**
  - System.Traceback.Symbolic

Quiz

What will be printed? (1/5)

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

What will be printed? (2/5)

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
         raise;
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

What will be printed? (3/5)

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A : Positive := -1;
   begin
      A := -5;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

What will be printed? (4/5)

```
with Text_IO; use Text_IO;
procedure E is
begin
   declare
      A, B, C : Positive;
   begin
      A := 10;
      B := 9;
      C := 2;
      A := B - A + C;
   exception
      when Constraint_Error =>
         Put_Line ("caught it");
   end;
exception
   when others =>
      Put_Line ("last chance handler");
end;
```

What is the assignment result? (5/5)

```
A, B : Integer := 5;

...

B := (if A /= 0 or raise Division_Error then B / A else 0);
```

Genericity

Barnes chapter 19

The notion of a pattern

- Sometimes, algorithms can be abstracted from the types that they operate on

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

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

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

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

Solution: generics

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

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

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

procedure Swap_I is new Swap (Integer);
procedure Swap_F is new Swap (Float);

I1, I2 : Integer;
F1, F2 : Float;

procedure Main is
begin
   Swap_I (I1, I2);
   Swap_F (F1, F2);
end Main;
```

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

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

int I1, I2;
float F1, F2;

void Main (void) {
   Swap <int> (I1, I2);
   Swap <float> (F1, F2);
}
```

What can be made generic?

- Subprograms & packages can be made generic
- Children of generic units have to be generic themselves

```
generic
   type T is private;
package Parent is […]

generic
package Parent.Child is […]

package I is new Parent (Integer);
package I_Child is new I.Child;
```

What can be made generic?

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

```
   generic
      type T is private;
   package P is
      V : T;
   end P;

   package I1 is new P (Integer);
   package I2 is new P (Integer);

begin

   I1.V := 5;
   I2.V := 6;

   if I1.V /= I2.V then
      -- will go there
```

Generic types parameters

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

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

- The actual parameter must provide at least as many properties as the generic contract
- The usage in the generic has to follow the contract

```
generic
   type T (<>) is private;
procedure P (V : T);

procedure P (V : T)
is
   X1 : T := V; -- OK, we can constrain the object by initialization
   X2 : T;      -- Compilation error, there is no constraint for this object
begin […]


procedure P1 is new P (String);  -- OK, unconstrained objects are accepted
procedure P2 is new P (Integer); -- OK, the object is already constrained
```

Properties that can be expressed on generic types

- **private** – any definite (and non-limited) type
- **(<>) private** – allowed to be indefinite
- **(<>)** – any discrete (integer or enumeration)
- **range <>** – any signed integer
- **mod <>** – any modular integer
- **digits <>** – any float
- **array** – array type (needs index and components)
- **access** – access type (needs target)

```
generic
   type T is (<>);
function Add_One (V : T) return T is
begin
   return T'Succ (V);
end Add_One;

function Add_One_I is new Add_One (Integer);
function Add_One_C is new Add_One (Character);
```

Generic parameters can be built one on top of the other

- **Consistency is checked at compile-time**

```
generic
   type T is private;
   type Index is (<>);
   type Arr is array (Index range <>) of T;
procedure P;

type Int_Array is array (Character range <>) of Integer;

procedure P_String is new P
   (T     => Integer,
    Index => Character,
    Arr   => Int_Array);
```

Generic constants & variables parameters

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

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

V : Float;

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

Generic subprograms parameters

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

```
generic
   with procedure Callback;
procedure P;

procedure P is
begin
   Callback;
end P;

procedure Something;

procedure P_I is new P (Something);
```

- “is <>” – matching subprogram is taken by default
- “is null” – null subprogram is taken by default

```
generic
   with procedure Callback_1 is <>;
   with procedure Callback_2 is null;
procedure P;

procedure Callback_1;

procedure P_I is new P; -- Will take Callback_1 and null
```

Generic Child Units

- **A generic unit can only have generic children, even if they don't have any parameters**

```
generic
   type T is private;
package Lists is
   [...]
```

```
generic

package Lists.Utils is
   [...]
```

- **To use a generic child, the parent must be instantiated first**

```
package L is new Lists (Integer);
package U is new L.Utils;
```

Quiz

Is there a compilation error? (1/8)

```
generic
   type T is private;
package G is
   V : T;
end G;
```

```
with G; use G;

procedure P is
   package I is new G (Integer);
begin
   V := 0;
end P;
```

Is there a compilation error? (2/8)

```
generic
   type T is private;
package G is
   V : T;
end G;
```

```
with G;

procedure P is
   type My_Integer is new Integer;

   package I1 is new G (Integer);
   package I2 is new G (My_Integer);

   use I1, I2;
begin
   V := 0;
end P;
```

Is there a compilation error? (3/8)

```
generic
   type T is private;
package G is
   V : T;
end G;
```

```
with G;

procedure P is
   type My_Integer is new Integer;

   package I1 is new G (Integer);
   package I2 is new G (My_Integer);

   use I1;
begin
   V := 0;
end P;
```

Is there a compilation error? (4/8)

```
generic
   type T is private;
package G is

end G;

generic
package G.Child is
   V : T;
end G.Child;
```

```
with G;

procedure P is
   package I1 is new G (Integer);
begin
   I1.Child.V := 0;
end P;
```

Is there a compilation error? (5/8)

```
generic
   type T (<>) is private;
package G is
   V : T;
end G;
```

```
with G;

procedure P is
   package I1 is new G (Integer);
begin
   I1.V := 0;
end P;
```

Is there a compilation error? (6/8)

```
generic
   type T is private;
package G is
   V : T;
end G;
```

```
with G;

package P is
   type My_Type is private;

   package I1 is new G (My_Type);
private
   type My_Type is null record;
end P;
```

Is there a compilation error? (7/8)

```
generic
   type T is private;
procedure P;

type R is record
   null;
end record;

type A is access all R;

procedure I1 is new P (Integer);
procedure I2 is new P (Float);
procedure I3 is new P (Character);
procedure I4 is new P (String);
procedure I5 is new P (R);
procedure I6 is new P (A);
```

Is there a compilation error? (8/8)

```
generic
   type T (<>) is private;
procedure P;

type R is record
   null;
end record;

type A is access all R;

procedure I1 is new P (Integer);
procedure I2 is new P (Float);
procedure I3 is new P (Character);
procedure I4 is new P (String);
procedure I5 is new P (R);
procedure I6 is new P (A);
```

Access Types

Barnes chapter 11

Access types design

- Java references, or C/C++ pointers are called access type in Ada
- An object is associated to a pool of memory
- Different pools may have different allocation / deallocation policies
- Without doing unchecked deallocations, and by using pool-specific access types, access values are guaranteed to be always meaningful
- In Ada, access types are <u>typed</u>

```
type Integer_Access is access Integer;

V : Integer_Access := new Integer;
```

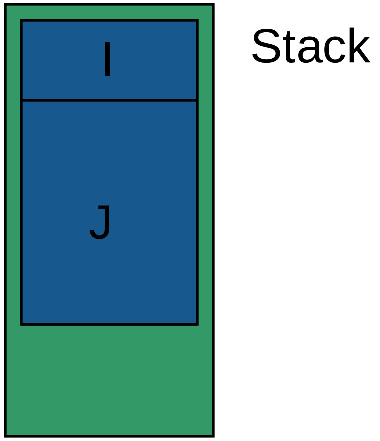
```
int * V = malloc (sizeof (int));

/* or in C++ */

int * V = new int;
```

Access types are dangerous

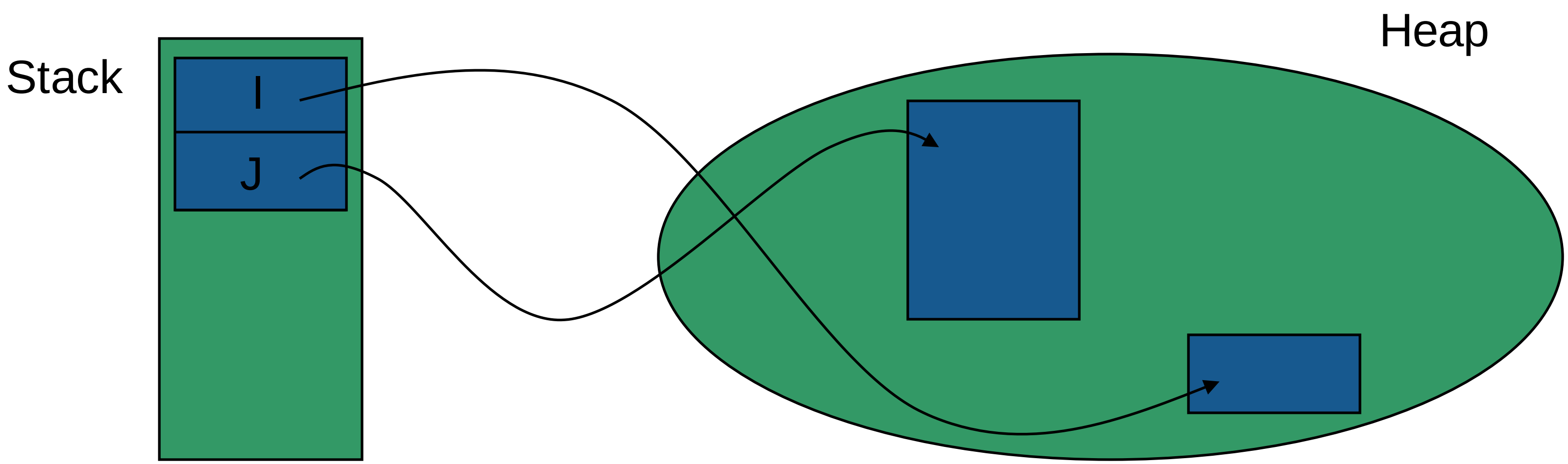
- **Multiple memory issues**
  - Leaks / corruptions
- **Introduces potential random failures complicated to analyze**
- **Increase the complexity of the data structures**
- **May decrease the performances of the application**
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more expensive than stacking objects
- **Ada avoids to use 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_Integer := new Integer'(0);
J : Access_String := new String'("Some Long String");
```



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 needed to move an object pointed by one type to another (pools may differ)**

- **You can not do this kind of conversion with a pool-specific access type:**

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

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);
```

Declaration location

- **Can be at library level**

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

- **Can be nested in a procedure**

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

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

Null values

- **A pointer that does not point to any actual data has a null value**
- **Without an initialization, a pointer is null by default**
- **null can be used in assignments and comparisons**

```
   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 introduces a leak
```

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 <u>lose</u> the safety of your pointers
- **But sometimes, you have to do what you have to do …**
  - There’s no simple way of doing it
  - Ada provides Ada.Unchecked_Deallocation
  - Has to be instantiated (it’s a generic)
  - Must work on an object, reset to null afterwards

Deallocation example

dependency on the generic subprogram

```
with Ada.Unchecked_Deallocation;

procedure P is
   type An_Access is access all A_Type;

   procedure Free is new Ada.Unchecked_Deallocation (A_Type, An_Access);

   V : An_Access := new A_Type;
begin
   Free (V);

end P;
```

V is null after the call

creation of the deallocation function (instance of the generic subprogram)

mention first the type and then the access.

Dereferencing pointers

- ***.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 all Integer;
type A_String is access all String;
type A_R is access all 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;
```

Pointing on objects declared on the stack

- **By default:**
  you cannot point to objects from the stack
  - What if the compiler has optimized the object to a register ?

- **Stack Objects on which an access can be created:**
  - Must be declared aliased
  - Accesses are then obtained through the *'Access* attribute
  - Only general pointers (declared with all) can point to such objects
  - Should not be deallocated
  - You should not keep references outside the scope of an object

Aliased objects example (1/2)

```
   type Acc is access all Integer;
   V : Acc;
   I : aliased Integer;
begin
   V := I'Access;
   V.all := 5; -- Same a I := 5
```

Aliased objects example (2/2)

```
type Acc is access all Integer;

G : Acc;

procedure P1 is
   I : aliased Integer;
begin
   G := I'Unchecked_Access; --  Same as 'Access (see after)
end P1;

procedure P2 is
begin
   G.all := 5; --  What if P2 is called after P1 ???
end P2;
```

Introduction to accessibility checks (1/2)

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

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

- **Access types can access to objects at most of the same depth**
- **The compiler checks it statically**
  (Removing checks is a workaround!)

Introduction to accessibility checks (2/2)

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

   procedure Proc is
      type T1 is access all Integer;
      A1 : T1;
      V1 : aliased Integer;
   begin
      A0 := V0'Access;
      A0 := V1'Access;
      A0 := V1'Unchecked_Access;

      A1 := V0'Access;
      A1 := V1'Access;
      A1 := T1 (A0);
      A0 := T0 (A1);

      A1 := new Integer;
      A0 := T0 (A1);
  end Proc;
end P;
```

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

Using pointers to create 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;
```

Common memory problems – uninitialized pointers

```
   type An_Access is access all Integer;

   V : An_Access;
begin
   V.all := 5;
```

Will raise Constraint_Error

Common memory problems – double deallocation

```
   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 the memory is still protected (deallocated)

May deallocate an other object if the memory has been reallocated – putting an object in an inconsistent state

Common memory problems – accessing deallocated memory

```
   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 the memory is still protected (deallocated)

May change an other object if the memory has been reallocated – putting an object in an inconsistent state

Common memory problems – memory leaks

```
    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…

Quiz

Is there an error? (1/10)

```
type An_Access is access all Integer;

W : Integer;
V : An_Access := W'Access;
```

Is there an error? (2/10)

```
type An_Access is access Integer;

W : aliased Integer;
V : An_Access := W'Access;
```

Is there an error? (3/10)

```
type An_Access is access all Integer;

procedure Proc is
   W : aliased Integer;
   X : An_Access := W'Access;
begin
   null;
end Proc;
```

Is there an error? (4/10)

```
type R is record
   F1, F2 : Integer;
end record;

type R_Access is access all R;

procedure Proc is
   V : R_Access := new R;
begin
   V.F1 := 0;
   V.all.F2 := 0;
end Proc;
```

Is there an error? (5/10)

```
G : aliased Integer;

procedure Proc is
   type A_Access is access all Integer;
   V : A_Access;
begin
   V := G'Access;
end Proc;
```

Is there an error? (6/10)

```
   type R is record
      F1, F2, F3 : Integer;
   end record;

   type R_Access is access all R;
   type R_Access_Access is access all R_Access;

   V : R_Access_Access;
begin
   V := new R_Access;
   V.all := new R;
   V.F1 := 0;
   V.all.F2 := 0;
   V.all.all.F3 := 0;
```

Is there an error? (7/10)

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V1 : A_Access := new Integer;
   V2 : A_Access := V1;
begin
   Free (V1);
   Free (V2);
```

Is there an error? (8/10)

```
    type A_Access is access all Integer;

    procedure Free is new Ada.Unchecked_Deallocation
       (Integer, A_Access);

    V : A_Access;
begin
    Free (V);
    V := new Integer;
    Free (V);
    Free (V);
```

Is there an error? (9/10)

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V : A_Access;
   W : aliased Integer;
begin
   V := W'Access;
   Free (V);
```

Is there an error? (10/10)

```
type A_Access is access all Integer;

type R is record
   V : A_Access;
   W : aliased Integer;
end record;

G : R;

procedure P is
   L : R;
begin
   G.V := G.W'Access;
   L.V := L.W'Access;
end P;
```

Inheritance

Barnes chapter 14

Primitives

The notion of a primitive

- **A type is characterized by two sets of properties**
  - Its data structure
  - The set of operations that applies to it
- **These operations are called “methods” in C++, or “Primitive Operations” in Ada**
- **In Ada**
  - the primitive relationship is implicit
  - The “hidden” parameter “this” is explicit (and can have any name)

```ada
type T is record
   Attribute_Data : Integer;
end record;

procedure Attribute_Function (This : T);
```

```cpp
class T {
   public:
      int Attribute_Data;
      void Attribute_Function (void);
};
```

General rule for a primitive

- **A subprogram S is a primitive of type T if**
  - S is declared in the scope of T
  - S has at least one parameter of type T (of any mode, including access) or returns a value of type T

```
package P is

   type T is range 1 .. 10;

   procedure P1 (V : T);
   procedure P2 (V1 : Integer; V2 : T);
   function F return T;

end P;
```

- **A subprogram can be a primitive of several types**

```
package P is

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

   procedure Proc (V1 : T1; V2 : T2);

end P;
```

Beware of access types !

- **Using a named access type in a subprogram creates a primitive of the <u>access type</u>, <u>NOT</u> the type of the accessed object!**

```
package P is

   type T is range 1 .. 10;
   type A_T is access all T;

   procedure Proc (V : A_T); -- Primitive of A_T

end P;
```

- **In order to create a primitive using an access type, the access mode should be used**

```
package P is

   type T is range 1 .. 10;
   procedure Proc (V : access T); -- Primitive of T

end P;
```

Implicit primitive operations

- **At type declaration, primitives are implicitly created if not explicitly given by the developer, depending on the kind of the type**

```
package P is

   type T1 is range 1 .. 10;
   --  implicitly declares function "+" (Left, Right : T1) return T1;
   --  implicitly declares function "-" (Left, Right : T1) return T1;
   --  …

   type T2 is null record;
   --  implicitly declares function "=" (Left, Right : T2) return T2;

end P;
```

- **These primitives can be used just as any others**

```
procedure Main is
   V1, V2 : P.T1;
begin
   V1 := P."+" (V1, V2);
end Main;
```

Use clauses

- Often, to avoid ambiguity and confusing overloading, “use package clauses” are forbidden by a coding standard. This means that all operations have to be prefixed, thus:

```
package A.B.C is
   type T1 is range 1 .. 10;
   procedure Print (V : T1);
end A.B.C;
```

```
with A.B.C;

procedure Main is
   V1, V2 : A.B.C.T1;
begin
   V1 := A.B.C."+" (V1, V2);
   A.B.C.Print (V1);
end Main;
```

- This is very annoying, though. I would prefer to write “V1 := V1 + V2” in the natural way…

Simple derivation

Simple type derivation

- **In Ada, any (non-tagged) type can be derived**

```
Type Child is new Parent;
```

- **A child is a distinct type inheriting from:**
  - The data representation of the parent
  - The primitives of the parent

```
   type Parent is range 1 .. 10;
   procedure Prim (V : Parent);

   type Child is new Parent;
   --  implicit procedure Prim (V : Child);

   V : Child;
begin
   V := 5;
   Prim (V);
```

- **Conversions are possible for non-primitive operations**

```
package P is
   type Parent is range 1 .. 10;
   type Child is new Parent;
end P;
```

```
procedure Main is
   procedure Not_A_Primitive (V : Parent);

   V1 : Parent;
   V2 : Child;
begin
   Not_A_Primitive (V1);
   Not_A_Primitive (Parent (V2));
end Main;
```

What can simple derivation do to the structure?

- **The structure of the type has to be kept**
  - An array stays an array
  - A scalar stays a scalar
- **Scalar ranges can be reduced**

```
type Int is range -100 .. 100;
type Nat is new Int range 0 .. 100;
type Pos is new Nat range 1 .. 100;
```

- **Constraints on unconstrained types can be specified**

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

type Ten_Elem_Arr is new Arr (1 .. 10);

type Rec (Size : Integer) is record
   Elem : Arr (1 .. Size);
end record;

type Ten_Elem_Rec is new Rec (10);
```

Signed Integer Types (revisited...)

Signed Integer Types (revisited)

- The "Basic Types" lecture introduced Ada's *signed integer* types, and the predefined Integer types in package Standard.

- But...we missed one important detail.

- A declaration like this:

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

- Is actually a short-hand for:

```
type <Anon> is new Predefined-Integer-Type;
subtype T is <Anon> range L .. R;
```

Signed Integer Types (revisited)

- **What's going on?**

```
type <Anon> is new Predefined-Integer-Type;
subtype T is <Anon> range L .. R;
```

1. **The compiler looks at L and R (which must be static) and chooses a predefined signed Integer type from Standard (e.g. Integer, Short_Integer, Long_Integer etc.) which at least includes the range L .. R.**
2. **This choice is *implementation-defined*.**
3. **An anonymous type <*Anon*> is created, derived from that predefined type. <*Anon*> inherits all of the predefined type's primitive operations, like "+", "-", "*" and so on.**
4. **A subtype T of <*Anon*> is created with range L .. R**

- **<*Anon*> can be referred to as T'Base in your program.**

Signed Integer Types (revisited)

- **What's going on?**

```
type <Anon> is new Predefined-Integer-Type;
subtype T is <Anon> range L .. R;
```

- **Warning! The choice of T'Base affects whether runtime computations will overflow.**
  - Example: on one machine, the compiler chooses Integer, which is 32-bit, and your code runs fine with no overflows.
  - On another machine, a compiler might choose Short_Integer, which is 16-bit, and your code will fail an Overflow_Check.
  - Extra care is needed if you have two compilers – e.g. for Host (like Windows or Linux) and Cross targets...

- **Good news! GNAT makes consistent and predictable choices on all major platforms.**

Signed Integer Types (revisited)

- **Guidance**
- **You can avoid the implementation-defined choice by deriving your own Base Types explicitly, and using Assert to enforce the expected range**
- **Something like**

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

- **Then derive further types and subtypes from My_Base_Integer**
- **Don't assume that "Shorter = Faster" for integer maths. On some machines, 32-bit is more efficient than 8- or 16-bit maths!**

Signed Integer Types (revisited)

- **Guidance 2**
- **If you want to derive from a base type that has a well-defined bit length (for example when dealing with hardware registers that *must* be a particular bit length), then package *Interfaces* declares types such as:**

```
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...
```

Quiz

Is there a compilation error? (1/10)

```
package P1 is

   type T1 is range 1 .. 10;

end P1;
```

```
with P1; use P1;

package P2 is

   type T2 is new T1;

end P2;
```

```
with P1; use P1;

package P3 is

   procedure Proc (V : T1);

end P3;
```

```
with P1; use P1;
with P2; use P2;
with P3; use P3;

procedure Main is
   V : T2;
begin
   Proc (V);
end Main;
```

Is there a compilation error? (2/10)

```
package P1 is

   type T1 is range 1 .. 10;
   procedure Proc (V : T1);

end P1;
```

```
with P1; use P1;

package P2 is

   type T2 is new T1;

end P2;
```

```
with P1; use P1;
with P2; use P2;

procedure Main is
   V : T2;
begin
   Proc (V);
end Main;
```

What’s the result of this call? (3/10)

```
package P is

   type T1 is range 1 .. 10;
   procedure Proc (V : T1);

   type T2 is range 1 .. 10;
   procedure Proc (V : T2);

end P;
```

```
with Ada.Text_IO; use Ada.Text_IO;

package body P is

   procedure Proc (V : T1) is
   begin
      Put_Line (“1”);
   end Proc;

   procedure Proc (V : T2) is
   begin
      Put_Line (“2”);
   end Proc;

end P;
```

```
with P; use P;

procedure Main is
   V1 : T1;
   V2 : T2;
begin
   Proc (V1);
   Proc (V2);
   Proc (T2 (V1));
   Proc (T1 (V2));
end Main;
```

Elaboration

Barnes chapter 13

Why elaboration is needed

- **Ada has some powerful features that require initialization:**

```
with Dep1;
package P1 is
   Val : constant Integer := Dep1.Call;
end P1;
```

Value is not known by the compiler

- **May also involve dynamic allocation:**

```
with P1;
package P2 is
   Buffer : String (1 .. P1.Val);
end P1;
```

Size is not known by the compiler

- **Or explicit user code to initialize a package**

```
package body P3 is
   …
begin
   Put_Line ("Starting P3");
end P3;
```

- **Requires initialization code at startup**
- **Implies ordering**

Elaboration

- **Process where entities are created**
- **The Rule: "an entity has to be elaborated before use"**
  - Subprograms have to be elaborated before being called
  - Variables have to be elaborated before being referenced
- **Such elaboration issues typically arise on**
  - Global variable initialization
  - Package sequence of statements

```
with Dep1;
package P1 is
   V_Spec : Integer := Dep1.Call;
   -- Dep1 body has to be elaborated before this point
end P1;

with Dep2;
package body P1 is
   V_Body : Integer;
begin
   V_Body := Dep2.Call;
   -- Dep2 body has to be elaborated before this point
end P1;
```

Elaboration order

- **The elaboration order is the order in which the packages are created**
- **It may or may not be deterministic**

```
package P1 is
   V_Spec : Integer := Call;
end P1;

package body P1 is
   V_Body : Integer := Call;
end P1;
```

```
package P2 is
   V_Spec : Integer := Call;
end P1;

package body P2 is
   V_Body : Integer := Call;
end P1;
```

- **The *binder (GNAT: gnatbind)* is responsible for finding an elaboration order**
  - Computes the possible ones
  - Reports an error when no order is possible

Circular elaboration dependencies

- Although not explicitly specified by the with clauses, elaboration dependencies may exhibit circularities
- Sometimes, they are static

```
package P1 is
   function Call return Integer;
end P1;

with P2;
package body P1 is
   V_Body : Integer := P2.Call;
end P1;
```

```
package P2 is
   function Call return Integer;
end P2;

with P1;
package body P2 is
   V_Body : Integer := P1.Call;
end P2;
```

- Sometimes they are dynamic

```
with P2;
package body P1 is
   V_Body : Integer;
begin
   if Day mod 2 = 1 then
      V_Body := P2.Call;
   end if;
end P1;
```

```
with P1;
package body P2 is
   V_Body : Integer;
begin
   if Day mod 2 = 0 then
     V_Body := P1.Call;
   end if;
end P2;
```

GNAT Static Elaboration Model

- **By default, GNAT ensures elaboration safety**
  - It adds elaboration control pragma to statically ensure that elaboration is possible
  - Very safe, but…
  - Not fully Ada compliant (may reject some valid programs)
  - Highly recommended however (least surprising effect)
- **Performed by gnatbind**
  - Automatically called by a builder (gnatmake or gprbuild)
  - Reads ALI files from the closure
  - Generates b~xxx.ad[sb] or b__xxx.ad[sb] files
  - Contains elaboration and finalization procedures
  - Defines the entry point procedure, main().

Pragma Preelaborate

- **Adds restrictions on a unit to ease elaboration**
- **Elaboration without explicit execution of code**
  - No user initialization code
  - No calls to subprograms
  - Static values
  - Dependencies only on Preelaborate packages

```
package P1 is
   pragma Preelaborate;
   Var : Integer := 7;
end P1;
```

- **But compiler may generate elaboration code**

```
package P1 is
  pragma Preelaborate;
  type ptr is access String;
  v : ptr := new String'("hello");
end P1;
```

Pragma Pure

- **Adds restrictions on a unit to ease elaboration**
- **Preelaborate +**
  - No variable declaration
  - No allocators
  - No access type declaration
  - Dependencies only on Pure packages

```
package Ada.Numerics is
   pragma Pure;
   Argument_Error : exception;
   Pi : constant := 3.14…;
end Ada.Numerics;
```

- **But compiler may generate elaboration code**

```
package P2 is
   pragma Pure;
   Var : constant Array (1 .. 10 * 1024) of Integer := (others => 118);
end P2;
```

Pragma Elaborate_Body

- Forces the elaboration of a body just after a specification
- Forces a body to be present even if none is required
- Problem: it may introduce extra circularities

```
package P1 is
   pragma Elaborate_Body;
   function Call return Integer;
end P1;

with P2;
package body P1 is

end P1;
```

```
package P2 is
   pragma Elaborate_Body;
   function Call return Integer;
end P2;

with P1;
package body P2 is

end P2;
```

- Useful in the case where a variable declared in the specification is initialized in the body

Pragma Elaborate

- **Pragma Elaborate forces the elaboration of a dependency body**
- **It does not force the elaboration of transitive dependencies**

```
package P1 is
   function Call return Integer;
end P1;
```

```
package P2 is
   function Call return Integer;
end P1;

with P1;
package body P2 is
   function Call return Integer
   begin
      P1.Call;
   end Call;
end P2;
```

```
with P2;
pragma Elaborate (P2);

package body P3 is
   V : Integer;
begin
   V := P2.Call;
end P3;
```

Pragma Elaborate_All

- Pragma Elaborate forces the elaboration of a dependency body and all transitive dependencies
- May introduce unwanted cycles
- Safer than Elaborate

```
package P1 is
   function Call return Integer;
end P1;
```

```
package P2 is
   function Call return Integer;
end P2;

with P1;
package body P2 is
   function Call return Integer
   begin
      P1.Call;
   end Call;
end P2;
```

```
with P2;
pragma Elaborate_All (P2);

package body P3 is
   V : Integer;
begin
   V := P2.Call;
end P3;
```

Bottom line

- Elaboration is a difficult problem to deal with
- The binder tries to resolve it in a “safe way”
- If it can’t, it’s possible to manually place elaboration pragmas
- Better to avoid elaboration constraints as much as possible
- Use dynamic elaboration (gnat binder switch -E) as last resort
- See ‘Elaboration Order Handling in GNAT’ annex in GNAT Pro User’s Guide.

Quiz

Is there a compilation error? (1/2)

```
package P is
   function F return Integer;

   A : Integer := F;
end P;
```

Is there a compilation error? (2/2)

```
with P2;
pragma Elaborate_All (P2);

package P1 is

end P1;
```

```
package P2 is

end P2;
```

```
with P1;

package body P2 is

end P2;
```

Tasking

Overview

A simple task

- Ada implements the notion of a “thread” via the task entity

```
procedure Main is
   task T;

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

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

Interacting with tasks

- **Active synchronization**
  - Client/server model of interaction (*"asymmetric rendezvous")*
  - Server task declares "*entries*" for interacting
    - Services it offers to other tasks
    - Can wait for a client task to request its service
  - Client task makes an "*entry call*"
    - Request for a service offered by another task
    - Will wait for the server task to "*accept*" and handle entry call
- **Passive synchronization**
  - Uses data objects with concurrency-safe access semantics
  - "*Protected objects*" in Ada – more about them later

Rendezvous (1/2)

- A task can declare "entries" for interacting and wait for an "entry call" to arrive

```
task T is
   entry Start;
   entry Receive_Message (V : String);
end T;

task body T is
begin
   loop
      accept Start;
      accept Receive_Message (V : String);
   end loop;
end T;
```

- When reaching an accept statement, the task will wait until its entry is called

- When calling an entry, the caller waits until the task is ready to be called

```
--  OK
T.Start;
T.Receive_Message ("");

--  Locks until somebody calls Start
T.Receive_Message ("");
```

Rendezvous (2/2)

- The task can perform operations while the caller and the callee are in the entry / accept statement

```
task T is
   entry Start;
   entry Receive_Message (V : String);
end T;

task body T is
begin
   loop
      accept Start do
         Put_Line ("Start");
      end Start;

      accept Receive_Message (V : String) do
         Put_Line ("Message : " & V);
      end Receive_Message;
   end loop;
end T;
```

- The caller will be released once the end of the accept block is reached

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**
  - 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

Protected objects

- Tasks are "active" objects
- Synchronization can be achieved through "passive" objects that hold and manage values
- A protected object is an object with an interface
  - No concurrent modifications are allowed
- It is a natural replacement for a lot of cases where a semaphore is needed

```
protected O is
   -- Only subprograms are allowed here
   procedure Set (V : Integer);
   function Get return Integer;
private
   -- Data declaration
   Local : Integer;
end O;
```

```
protected body O is
   procedure Set (V : Integer) is
   begin
      Local := V;
   end Set;

   function Get return Integer is
   begin
      return Local;
   end Get;
end O;
```

Protected functions vs. protected procedures

- **Procedures can modify the state of the protected data**
  - No concurrent access to procedures can be done
  - No procedure can be called when functions are called
- **Functions are just ways to retrieve values, the protected data is read-only**
  - Concurrent access to functions can be done
  - No function can be called when a procedure is called

Task types

- **It is possible to create task types**
  - Objects can be instantiated on the stack or on the heap
- **Tasks instantiated on the stack are activated at the end of the elaboration of their enclosing declarative part**
  - As if they were declared there
- **Tasks instantiated on the heap are activated right away**

```
task type T is
   entry Start;
end T;

type T_A is access all
T;

task body T is
begin
   accept Start;
end T;
```

```
   V1 : T;
   V2 : T;
   V3 : A_T;
begin
   V1.Start;
   V2.Start;
   V3 := new T;
   V3.all.Start;
```

- **Tasks are limited objects (no copies allowed)**

Protected object types

- Like tasks, protected objects can be defined through types
- Instantiation can then be done on the heap or the stack
- Protected object types are limited types

```
protected type O is
   entry Push (V : Integer);
   entry Pop  (V : out Integer);
private
   Buffer : Integer_Array (1 .. 10);
   Size : Integer := 0;
end O;

type O_Access is access all O;
```

```
V1, V2 : O;
V3 : O_Access := new O;
```

```
protected body O is
   entry Push (V : Integer)
      when Size < Buffer'Length
   is
   begin
      Buffer (Size + 1) := V;
      Size := Size + 1;
   end Push;

   entry Pop  (V : out Integer)
      when Size > 0
   is
   begin
      V := Buffer (Size);
      Size := Size - 1;
   end Pop;
end O;
```

Scope of a task

- **Tasks can be nested in any declarative block**
- **When nested in e.g. a subprogram, the task and the subprogram body have to finish before the subprogram ends**
- **Tasks declared at library level all have to finish before the program terminates**

```
package P is
   task T;
end P;

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

Some Advanced Concepts…

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, and accept the first one that is requested

```
task T is
   entry Start;
   entry Receive_Message (V : String);
   entry Stop;
end T;

task body T is
begin
   accept Start;
   loop
      select
         accept Receive_Message (V : String) do
            Put_Line ("Message : " & String);
         end Receive_Message;
      or
         accept Stop;
         exit;
      end select;
   end loop;
end T;
```

```
T.Start;
T.Receive_Message ("A");
T.Receive_Message ("B");
T.Stop;
```

Waiting with a delay

- A **select** statement can wait for only a given amount of time, and then do something when that **delay** is exceeded

```
task T is
   entry Receive_Message (V : String);
end T;

task body T is
begin
   loop
      select
         accept Receive_Message (V : String) do
            Put_Line ("Message : " & String);
         end Receive_Message;
      or
         delay 50.0;
         Put_Line ("Don't wait any longer");
         exit;
      end select;
   end loop;
end T;
```

- the "**delay until**" statement can be used as well
- there can be multiple delay statements (useful when the value is not hard-coded)

Calling an entry with a delay protection

- **A call to an entry normally blocks the thread until the entry can be accepted by the task**
- **It is possible to wait for a given amount of time using a select … delay statement**

```
task T is
   entry Receive_Message (V : String);
end T;

procedure Main is
begin
   select
      T.Receive_Message ("A");
   or
      delay 50.0;
   end select;
end Main;
```

- **Only one entry call is allowed**
- **No "accept statement" is allowed**

Avoid waiting if no entry or accept can be taken

- The “else” part allows to avoid waiting if the accept statements or entries are not ready to be entered
- No delay statement is allowed in this case

```
task T is
   entry Receive_Message (V : String);
end T;

task body T is
begin
   select
      accept Receive_Message (V : String) do
         Put_Line ("Received : " & V);
      end Receive_Message;
   else
      Put_Line ("Nothing to receive");
   end select;
end T;


procedure Main is
begin
   select
      T.Receive_Message ("A");
   else
      Put_Line ("Receive message not called");
   end select;
end Main;
```

Terminate alternative

- **When waiting for an entry, if all other task dependent on the same master task (including the master task) are terminated, the entry can't be called anymore**
- **This can be detected by the "or terminate" alternative, which terminates the tasks if all other tasks are terminated**
  - Or themselves waiting on "or terminate" select statements
- **Once reached, the task is terminated right away, no additional code is called**

```
select
   accept E;
or
   terminate;
end select;
```

Guard expressions

- **The accept statement can be activated according to a guard condition**
- **This condition is evaluated when entering select**

```
task T is
   entry Put (V : Integer);
   entry Get (V : out Integer);
end T;

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 (1/2)

- **Protected entries are a special kind of protected procedures**
- **They can be defined using a barrier, a conditional expression allowing the entry to be called or not**
- **The barriers are evaluated…**
  - Every time a task request to call an entry
  - Every time a protected entry or procedure is exited

```
protected O is
   entry Push (V : Integer);
   entry Pop  (V : out Integer);
private
   Buffer : Integer_Array (1 .. 10);
   Size : Integer := 0;
end O;
```

```
protected body O is
   entry Push (V : Integer)
      when Size < Buffer'Length
   is
   begin
      Buffer (Size + 1) := V;
      Size := Size + 1;
   end Push;

   entry Pop  (V : out Integer)
      when Size > 0
   is
   begin
      V := Buffer (Size);
      Size := Size - 1;
   end Pop;
end O;
```

Protected object entries (2/2)

- **Several tasks can be waiting on entries**
- **Only one task is reactivated when the barrier is relieved, depending on the activation policy**

```
task body T1 is
   V : Integer;
begin
   O.Pop (V);
end T1;

task body T2 is
   V : Integer;
begin
   O.Pop (V);
end T2;

task body T3 is
begin
   delay 1.0;
   O.Push (42);
end T3;
```

Select on protected objects entries

- **Works the same way as select on task entries**
  - With a delay part

```
select
   O.Push (5);
or
   delay 10.0;
   Put_Line ("Delayed overflow");
end select;
```

  - With an else part

```
select
   O.Push (5);
else
   Put_Line ("Overflow");
end select;
```

Notion of a Queue

- Protected entries, protected procedures and task entries can only be activated by one task at a time
- If several tasks are trying to enter a mutually exclusion section, they are put in a queue
- By default, task are entering the queue in FIFO
- If several tasks are in a queue when the server task is terminated, TASKING_ERROR is sent to the waiting tasks

Requeue instruction

- The “**requeue**” instruction can be called in an entry (task or protected)
- It places the queued task back to another entry with the same profile
  - Or the same entry…
- Useful if the treatment couldn’t be done and need to be re-considered 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

Abort Statements

- All tasks can be abruptly aborted

```
procedure Main is
   task T;

   task T is
   begin
      loop
         delay 1.0;
         Put_Line ("A");
      end loop;
   end T;
begin
   delay 10.0;
   abort T;
end;
```

- Abortion may stop the task almost anywhere in the assembly code
- Highly unsafe – should be used only as last resort

Quiz

Does this code compile? (1/8)

```
protected O is
   function Get return Integer;

   procedure Set (V : Integer);
private
   Val : Integer;
   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
      Val := V;
   end Set;
end O;
```

What is the output of this code? (2/8)

```
procedure Main is
   task T is
      entry A;
   end T;

   task body T is
   begin
      select
         accept A;
      or
         terminate;
      end select;

      Put_Line ("Terminated");
   end T;
begin
   null;
end Main;
```

What is the output of this code? (3/8)

```
procedure Main is
begin
   select
      delay 2.0;
   then abort
      loop
         delay 1.5;
         Put_Line ("A");
      end loop;
   end select;
   Put_Line ("B");
end Main;
```

Does this code compile? (4/8)

```
task T is
   entry Remove_Items (Nb : Integer);
   entry Replenish;
end T;

task body T is
   Nb_Items : Integer := 100;
begin
   loop
      select
         accept Remove_Items (Nb : Integer) do
            if Nb_Items < Nb then
               requeue Replenish;
            else
               Nb_Items := Nb_Items - Nb;
            end if;
         end Remove_Items;
      or
         accept Replenish do
            Nb_Items := Nb_Items + 100;
         end Replenish;
      end select;
   end loop;
end T;
```

What’s the output of this code? (5/8)

```
task body T1 is
begin
   loop
      select
         accept A;
         Put_Line ("SELECT TASK");
      else
         delay 1.0;
         Put_Line ("ELSE TASK");
      end select;
   end loop;
end T1;
```

```
task body T2
begin
   loop
      select
         T1.A;
       else
         delay 1.0;
      end select;
   end loop;
end T2;
```

Does this code compile? (6/8)

```
task T1 is
   entry E1;
   entry E2;
end T1;
```

```
task body T2
begin
   select
      T1.E1;
   or
      T1.E2;
   end select;
end T2;
```

Does this code terminate? (7/8)

```
procedure Main is
   Ok : Boolean := False;

   protected O is
      entry P;
   end O;

   protected body O is
   begin
      entry P when Ok is
         Put_Line ("OK");
      end P;
   end O;

   task T;

   task body T is
   begin
      delay 1.0;
      Ok := True;
   end T;
begin
   O.P;
end;
```

Does this code terminate? (8/8)

```
procedure Main is
   Ok : Boolean := False;

   protected O is
      entry P;
      procedure P2;
   end O;

   protected body O is
      entry P when Ok is
      begin
         Put_Line ("OK");
      end P;

      procedure P2 is
      begin
         null;
      end P2;
   end O;

   task T;

   task body T is
   begin
      delay 1.0;
      Ok := True;
      O.P2;
   end T;
begin
   O.P;
end;
```