AdaCore
The GNAT Pro Company

Le langage Ada

Tristan Gingold

gingold@adacore.com

Historique

- **Dans les années 70, le DOD souffre d'une explosion du nombre des langages utilisés**
- **Il lance un concours international pour un langage qui répond à toutes ces exigences (1974):**
- **Plusieurs itérations**
- **Vainqueur:**
  - L'équipe de Jean Ichbiah, CII Honeywell Bull
- **Première normalisation du langage (ANSI, ISO): 1983**
- **Révisions majeures en 1995, 2005, 2012.**
- **L'un des seuls langages normalisés *à priori***

Exigences du language

- **Généraliste**
  - Efficacité
  - Simplicité
  - Implémentabilité
- **Haut niveau de génie logiciel**
  - Maintenabilité
  - Portabilité
  - Fiabilité
- **Norme claire et non ambiguë**
- **Travail sur des plateformes embarquées**
- **Traitements parallèles**
- **Gestion des données bas niveau**

Resultat : Ada

- **Dérivé d'une syntaxe type Pascal**
- **Impératif**
  - Dans la même classe que Fortran, Cobol, C/C++, Java, Python…
- **Parallélisme intégré au langage (par opposition aux API type pthread)**
- **Modulable (facilité de mise en place de sous-ensembles)**
- **Vérifications statiques et dynamiques (bornes…)**

Ada aujourd'hui

- **Marché privilégié:**
  - Systèmes temps-réel
  - Systèmes critiques (safety critical, mission critical)
  - Systèmes de sécurité (MILS)

- **Exemples**
  - 787 Dreamliner (Common Core System)
  - Airbus A350 XWB (Air Data Inertial Reference Unit)
  - Sentinel 1 (Environmental Satellite System)
  - Canadian Space Arm
  - Meteor (metro line 14)
  - ...

Example: "Hello, World" Program

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

--  Display a welcome message
procedure Greet is
begin
   Put_Line("Hello, World!");
end Greet;
```

A comment

A statement: procedure call

```
$ gnatmake greet.adb
$ ./greet
```

AdaCore
The GNAT Pro Company

Imperative language

For loops

Iterator.
Like a constant,
declared only in the loop

Range.
Usually low .. high.

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

procedure Greet is
begin
   for I in 1 .. 10 loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

Procedure call.
Cannot call a function
without using its value.

While loops

Declaration of a variable

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

procedure Greet is
  I : Integer := 1;
begin
   while I < 10 loop
      Put_Line("Hello, World!");
      I := I + 1;
   end loop;
end Greet;
```

Condition.
Must be of type Boolean

Assignment.

General loops

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

procedure Greet is
  I : Integer := 1;
begin
   loop
      Put_Line("Hello, World!");
      exit when I = 5;
      I := I + 1;
   end loop;
end Greet;
```

Condition.

Exit statement

If

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

procedure Greet is
  I : Integer := 1;
begin
   loop
      Put_Line("Hello, World!");
      if I = 5 then
        exit;
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Condition.

If/else

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

procedure Greet is
  I : Integer := 1;
begin
   loop
      if I = 5 then
        exit;
      else
        Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

If/elsif/else

No need to nest the if

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

procedure Greet is
  I : Integer := 0;
begin
   loop
      if I = 5 then
        exit;
      elsif I = 0 then
        Put_Line (“Starting...”);
      else
        Put_Line ("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

If/elsif/else

When a statement is expected, you should provide one. The null statement does nothing.

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

procedure Greet is
  I : Integer := 0;
begin
   loop
      if I = 5 then
        exit;
      elsif I = 0 then
        null;
      else
        Put_Line ("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

If/else

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

procedure Greet is
  I : Integer := 1;
begin
   loop
      if I = 5 or else I = 2 then
        exit;
      else
        Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Short-circuit or

If/else

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

procedure Greet is
  I : Integer := 1;
begin
   loop
      if I = 5 and then I = 2 then
        exit;
      else
        Put_Line("Hello, World!");
      end if;
      I := I + 1;
   end loop;
end Greet;
```

Short-circuit and

case

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

procedure Greet is
  I : Integer := 0;
begin
   loop
      case I is
         when 0 =>
           Put_Line (“Starting...”);
         when 3 .. 4 =>
           Put_Line (“Hello”);
         when 7 | 9 =>
           Put_Line (“Guten Tag”);
         when 12 =>
           exit;
         when others =>
           Put_Line ("Hello, World!");
      end case;
      I := I + 1;
   end loop;
end Greet;
```

Expression must be of a discrete type.

All the values must be covered.

‘when others’ must be the last one and alone (if present)

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
for I in 10 .. 1 loop
   Put_Line("Hello, World!");
end loop;
```

2: Is there a compilation error ?

```
for I in reverse 1 .. 10 loop
   Put_Line("Hello, World!");
end loop;
```

3: Is there a compilation error ?

```
procedure Hello is
  I : Integer;
begin
   for I in 1 .. 10 loop
      Put_Line ("Hello, World!");
   end loop;
end Hello;
```

4: Is there a compilation error ?

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

procedure Greet is
  I : Integer;
begin
   while I < 10 loop
      Put_Line("Hello, World!");
      I := I + 1;
   end loop;
end Greet;
```

5: Is there a compilation error ?

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

procedure Greet is
  I : Integer := 2;
begin
   while i < 10 loop
      Put_Line ("Hello, World!");
      i := i + 1;
   end loop;
end Greet;
```

6: Is there a compilation error ?

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

procedure Greet is
begin
   loop
      Put_Line("Hello, World!");
      Tools.My_Proc;
   end loop;
end Greet;
```

7: Is there a compilation error ?

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

procedure Greet is
  I : Integer := 0;
begin
   loop
      if I = 5 then
        exit;
      else
        if I = 0 then
          Put_Line (“Starting...”);
        else
          Put_Line ("Hello, World!");
        end if;
      end if;
      I := I + 1;
   end loop;
end Greet;
```

8: Is there a compilation error ?

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

procedure Greet is
  I : Integer := 0;
begin
   loop
      case I is
         when 0 =>
           Put_Line ("Starting...");
         when 1 .. 4 =>
           Put_Line ("Hello");
         when 5 =>
           exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

9: Is there a compilation error ?

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

procedure Greet is
begin
   loop
      case I is
         when 0 =>
           Put_Line ("Starting...");
         when 1 .. 4 =>
           Put_Line ("Hello");
         when others =>
           exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

10: Is there a compilation error ?

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

procedure Greet is
   I : Integer := 0;
begin
   loop
      case I is
         when Integer'First .. 1 =>
           Put_Line ("Starting...");
         when 1 .. 4 =>
           Put_Line ("Hello");
         when others =>
           exit;
      end case;
      I := I + 1;
   end loop;
end Greet;
```

11: Is there a compilation error ?

```
V : Integer;
```

```
1V : Integer;
```

```
V_ : Integer;
```

```
_V : Integer;
```

```
V__1 : Integer;
```

```
V_1 : Integer;
```

AdaCore
The GNAT Pro Company

Strongly typed language

Question

What is a type ?

Integers

Declare a signed type, and give the bounds

Range.
Usually low .. high,
but could be Arr'Range

Declarations can appear only in a declarative region.

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

procedure Greet is
  type Entier is range 1 .. 20;
begin
   for I in Entier loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

Integers

Get the higher bound of a type or a subtype.
Use ‘First for the lower bound.

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

procedure Greet is
  type Entier is range 1 .. 20;
begin
   for I in Entier loop
      if I = Entier’Last then
        Put_Line (“Bye”);
      else
        Put_Line("Hello, World!");
      end if;
   end loop;
end Greet;
```

Integers

Overflow.
Will raise an exception at run-time.
(use -gnato for old versions of GNAT)

```
procedure Greet is
  A : Integer := Integer'Last;
  B : Integer;
begin
   B := A + 5;
end Greet;
```

Integers

No overflow here.
Computation is done in the base type of Entier.

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

procedure Greet is
  type Entier is range 1 .. 20;
  A : Entier := 12;
  B : Entier := 15;
  M : Entier := (A + B) / 2;
begin
   for I in 1 .. M loop
      Put_Line("Hello, World!");
   end loop;
end Greet;
```

Enumerations

A enumeration type

Explicit list of values

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
begin
   for I in Days loop
      case I is
        when Saturday .. Sunday =>
          Put_Line ("Week end!");
        when others =>
          Put_Line (“Hello on “ & Days’Image (I));
       end case;
   end loop;
end Greet;
```

‘Image attribute
Function that convert a value to a string

Strong typing

Declare two signed types

Declare a constant

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

procedure Greet is
   type Meters is range 0 .. 10_000;
   type Miles is range 0 .. 5_000;
   Dist_Us : Miles;
   Dist_Eu : constant Meters := 100;
begin
   Dist_Us := Dist_Eu * 1609 / 1000;
   Put_Line (Miles'Image (Dist_Us));
end Greet;
```

Not correct: types mismatch

Easy way to display a value

Strong typing

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

procedure Conv is
  type Meters is range 0 .. 10_000;
  type Miles is range 0 .. 5_000;
  Dist_Us : Miles;
  Dist_Eu : constant Meters := 100;
begin
   Dist_Us := Miles (Dist_Eu * 1609 / 1000);
   Put_Line (Miles'Image (Dist_Us));
end;
```

Type conversion

Strong typing

Predefined type for characters.

Character literal

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

procedure Greet is
  C : Character;
begin
  C := '?';
  C := 64;
end Greet;
```

Not correct:
a number is not an enumeration literal
types mismatch

Strong typing

Use ”” to insert a quote in a string

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

procedure Greet is
  C : Character;
begin
  C := ‘?’;
  Put_Line (“”“Ascii”” code of ‘“ & C & “‘ is”
            & Integer’Image (Character’Pos (C)));
  C := Character’Val (64);
end Greet;
```

‘Val attribute convert a position to its value.

‘Pos attribute convert a value to its position.

Subtypes

Declare a subtype

Constraint of the subtype

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  subtype Weekend_Days is Days range Saturday .. Sunday;
begin
   for I in Days loop
      case I is
        when Weekend_Days =>
          Put_Line ("Week end!");
        when others =>
          Put_Line (“Hello on “ & Days’Image (I));
      end case;
   end loop;
end Greet;
```

A subtype can be used as a range

A subtype doesn't define a new type

Correct: same type

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  subtype Weekend_Days is Days range Saturday .. Sunday;
  Day : Days := Saturday;
  Weekend : Weekend_Days;
begin
   Weekend := Day;
   Weekend := Monday;
end Greet;
```

Correct at compile time, exception at run-time: constraint error

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
type Entier is range 1 .. 20.5;
```

2: Is there a compilation error ?

```
type Entier is range 1 .. 20.0;
```

3: Is there a compilation error ?

```
A : Integer := 5;
type Entier is range A .. 20;
```

4: Is there a compilation error ?

```
type Entier is range 1 .. Integer'Last;
```

5: Is there a compilation error ?

```
type Entier1 is range 1 .. Integer'Last;
type Entier2 is range Integer'First .. 0;
type Entier3 is range Entier2'First .. Entier1'Last;
```

6: Is there a compilation error ?

```
type Entier1 is range 1 .. Integer'Last;
subtype Entier2 is Entier1 range 1 .. 100;

V1 : Entier1 := 5;
V2 : Entier2;

V2 := V1;
```

7: Is there a compilation error ?

```
type Entier1 is range 1 .. Integer'Last;
type Entier2 is range 1 .. 100;

V1 : Entier1 := 5;
V2 : Entier2;

V2 := V1;
```

8: Is there a compilation error ?

```
type Enum is (E1, E2);
type Enum2 is (E2, E3);
```

9: Is there a compilation error ?

```
type Bit is ('0', '1');
```

AdaCore
The GNAT Pro Company

Arrays

Array type declaration (1/2)

Bounds of the array

Type of elements

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Index is range 1 .. 5;
  type Tableau is array (Index) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Index loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Array type declaration (2/2)

aggregate

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Index is range 1 .. 5;
  type Tableau is array (Index) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Index loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Array index

No predefined base

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Index is range 11 .. 15;
  type Tableau is array (Index) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Index loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Array index

Index type can be any discrete type

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

procedure Greet is
  type Entier is range 1 .. 31;
  type Month is (Jan, Feb, Mar, Apr, May, Jun,
                 Jul, Aug, Sep, Oct, Nov, Dec);
  type Tableau is array (Month) of Entier;
  Tab : constant Tableau := (31, 28, 31, 30, 31, 30,
                             31, 31, 30, 31, 30, 31);
begin
   for I in Month loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

A variable that cannot be modified

Indexation (1/2)

Indexes are checked: will raise an exception at run time.

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Index is range 1 .. 5;
  type Tableau is array (Index) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Index range 2 .. 6 loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Indexation

Indexation (2/2)

Compilation error: type of I is Integer, not Entier (strong typing).

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Index is range 1 .. 5;
  type Tableau is array (Index) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Natural range 1 .. 5 loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Shortcut for index

subtype of Integer

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Tableau is array (1 .. 5) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in 1 .. 5 loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Likewise

Index range

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

procedure Greet is
  type Entier is range 0 .. 1000;
  type Tableau is array (1 .. 5) of Entier;
  Tab : Tableau := (2, 3, 5, 7, 11);
begin
   for I in Tab'Range loop
     Put (Entier'Image (Tab (I)));
   end loop;
   New_Line;
end Greet;
```

Get the range of Tab

Predefined type: String

String is a predefined array type of Character.

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

procedure Greet is
  Message : String (1 .. 11) := “Hello World”;
begin
   for I in reverse 1 .. 11 loop
     Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

Iterate in reverse order

Arrays

The compiler can automatically compute the bounds from the initial value.

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

procedure Greet is
  Message : constant String := “Hello World”;
begin
   for I in reverse Message’First .. Message’Last loop
     Put (Message (I));
   end loop;
   New_Line;
end Greet;
```

‘First and ‘Last attributes on an array returns the low and high bound

Declaring arrays

Subtype of the String type

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  subtype Day_Name is String (1 .. 2);
  type Days_Name_Type is array (Days) of Day_Name;
begin
   null;
end Greet;
```

Type of the index

Type of the element.
Must be a definite type.

Declaring arrays

Initial value is given by an aggregate

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  subtype Day_Name is String (1 .. 2);
  type Days_Name_Type is array (Days) of Day_Name;
  Names : constant Days_Name_Type :=
    ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
begin
   for I in Names'Range loop
     Put_Line (Names (I));
   end loop;
end Greet;
```

Unconstrained arrays

Indefinite array type.
Bounds are not known

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  type WorkLoad_Type is array (Days range <>) of Natural;
  Workload : constant Workload_Type (Monday .. Friday) :=
    (Friday => 7, others => 8);
begin
   for I in Workload'Range loop
     Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Associate by name

‘Default’ value

Specify the bounds of the array.

Declaring arrays

Bounds can be deduced from the initialization value

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  type WorkLoad_Type is array (Days range <>) of Natural;
  Workload : constant Workload_Type :=
    (Monday .. Thursday => 8, Friday => 7);
begin
   for I in Workload'Range loop
     Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

Associate by name

Declaring arrays

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  type WorkLoad_Type is array (Days range <>) of Natural;
  Workload : constant Workload_Type :=
    (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);
begin
   for I in Workload'Range loop
     Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

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

2: Is there a compilation error ?

```
type Str_Array is array (1 .. 10) of String;
```

3: Is there a compilation error ?

```
A : constant Integer := 5;
```

4: Is there a compilation error ?

```
A : constant String (1 .. 12);
```

AdaCore
The GNAT Pro Company

Modular/Structured programming

Package

Only declarations,
no statements

```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  type WorkLoad_Type is array (Days range <>) of Natural;
  Workload : constant Workload_Type :=
    (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);
end Week;
```

Package

Group related declarations together

Define an interface (API)

Hide the implementation

Provide a name space

```
package Week is
…
end Week;
```

With-ing a package

Reference the package.
Add a dependency on that package.

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

procedure Greet is
begin
   for I in Workload'Range loop
     Put_Line (Integer'Image (Workload (I)));
   end loop;
end Greet;
```

With-ing a package

Makes declaration of Ada.Text_IO visible (In particular Put_Line).

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

procedure Greet is
begin
   for I in Week.Workload'Range loop
     Put_Line (Integer'Image (Week.Workload (I)));
   end loop;
end Greet;
```

Workload is not directly visible, but can be referenced by selection

Package body

Additional declarations.
Not visible outside the body

Package body

```
package body Week is
   type WorkLoad_Type is array (Days range <>) of Natural;
   Workload : constant Workload_Type :=
      (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);

   function Get_Workload (D : Days) return Natural is
   begin
      return Workload (D);
   end;
end Week;
```

Return statement
(Required for a function)

Body of the function.
Required for all specification in of the package.

AdaCore
The GNAT Pro Company

Subprograms

Subprograms

Declare and define a procedure, without parameters.

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

--  Display a welcome message
procedure Greet is
begin
   Put_Line("Hello, World!");
end Greet;
```

Subprograms

A function specification, with one parameter and result type.

```
package Week is
   type Days is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
   function Get_Workload (D : Days) return Natural;
end Week;
```

Subprograms

```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  function Get_Day_Name (D : Days := Monday) return String;
end Week;
```

Default value.

Any type can be returned, including indefinite one.

Subprograms

```
package body Week is
  function Get_Day_Name (D : Days := Monday) return String is
  begin
     case D is
        when Monday => return “Monday”;
        when Tuesday => return “Tuesday”;
        ...
        when Sunday => return “Sunday”;
     end case;
end Week;
```

Parameters mode

```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  procedure Next_Day (Res : out Days; D : Days);
end Week;
```

Mode out: not initialized at the beginning, procedure can assign it

No mode, default is ‘in’.
Like a constant within the body.

Parameters mode

Same type and mode for parameters A and B.

Mode in out: initialized at the beginning, procedure can assign it.
Like a variable.

```
procedure Swap (A, B : in out Integer)
is
   Tmp : Integer;
begin
   Tmp := A;
   A := B;
   B := Tmp;
   return;
end Swap;
```

Return statement.
Optional for procedures.

Subprogram call

Association by position

```
procedure Test_Swap
is
   X, Y : Integer;
begin
   X := 5;
   Y := 7;
   Swap (X, Y);
   Swap (A => X, B => Y);
   Swap (B => X, A => Y);
end Test_Swap;
```

Association by name.
Can makes the code more readable.

Mutually recursive subprograms

Subprogram specification

```
procedure Compute_A (V : Natural);

procedure Compute_B (V : Natural) is
begin
  if V > 5 then
    Compute_A (V - 1);
  end if;
end Compute_B;

procedure Compute_A (V : Natural) is
begin
  if V > 2 then
    Compute_B (V - 1);
  end if;
end Compute_A;
```

Nested subprograms

```
procedure Compute_B (V : Natural)
is
   procedure Compute_A is
   begin
     if V > 2 then
       Compute_B (V - 1);
     end if;
   end Compute_A;
begin
  if V > 5 then
    Compute_A;
  end if;
end Compute_B;
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
package My_Type is
  type My_Type is range 1 .. 100;
end My_Type;
```

2: Is there a compilation error ?

```
package Pkg is
  function f (A : Integer);
end Pkg;
```

3: Is there a compilation error ?

```
package Pkg is
  function f (A : Integer) return Integer;
  function f (A : Character) return Integer;
end Pkg;
```

4: Is there a compilation error ?

```
package Pkg is
  function f (A : Integer) return Integer;
  procedure f (A : Character);
end Pkg;
```

5: Is there a compilation error ?

```
package Pkg is
  subtype Int is Integer;
  function f (A : Integer) return Integer;
  function f (A : Int) return Integer;
end Pkg;
```

6: Is there a compilation error ?

```
package Pkg is
  procedure Proc (A : Integer);
  procedure Proc (A : in out Integer);
end Pkg;
```

7: Is there a compilation error ?

```
package Pkg is
  procedure Proc (A : in out Integer := 7);
end Pkg;
```

8: Is there a compilation error ?

```
package Pkg is
  procedure Proc (A : Integer := 7);
end Pkg;
```

```
package body Pkg is
  procedure Proc (A : Integer) is
    ...
  end Proc;
end Pkg;
```

9: Is there a compilation error ?

```
package Pkg is
  procedure Proc (A : in out Integer);
end Pkg;
```

```
package body Pkg is
  procedure Proc (A : in out Integer) is
    ...
  end Proc;

  procedure Proc (A : in out Character) is
    ...
  end Proc;
end Pkg;
```

10: Is there a compilation error ?

```
package Pkg is
  procedure Proc (A : in Integer);
end Pkg;
```

```
package body Pkg is
  procedure Proc (A : in Integer);

  procedure Proc (A : in Integer) is
    ...
  end Proc;
end Pkg;
```

11: Is there a compilation error ?

```
package Pkg1 is
   ...
end Pkg1;
```

```
with Pkg1;

package Pkg2 is
   ...
end Pkg2;
```

```
with pkg2;

...
   Pkg1.Proc
...
```

13: Is there a compilation error ?

```
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;
```

```
with Pkg1; use Pkg1;

package Pkg2 is
  ...
end Pkg2;
```

```
package body Pkg2 is
  ...
    Proc;
  ...
end Pkg2;
```

14: Is there a compilation error ?

```
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;
```

```
with Pkg1; use Pkg1;

package Pkg2 is
  ...
end Pkg2;
```

```
with Pkg1; use Pkg1;

package body Pkg2 is
  ...
end Pkg2;
```

15: Is there a compilation error ?

```
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;
```

```
with Pkg1;

package Pkg2 is
  ...
end Pkg2;
```

```
use Pkg1;

package body Pkg2 is
  ...
    Proc;
  ...
end Pkg2;
```

AdaCore
The GNAT Pro Company

More about types

Array

Size of the array is not known at compile time.
But bounds are fixed

```
Len : Natural := f (5);
Buf : String (1 .. Len);
...
Len := 3;
```

No side-effect on Buf.

Array: Slice

```
Buf : String (1 .. 12);
...
Buf (2 .. 4) := “Abc”;
```

A range

Records

```
type Date is record
  Day : Integer range 1 .. 31;
  Month : Month_Name;
  Year : Integer range 1 .. 3000;
end record;
```

Components

Records - Default values

```
type Date is record
  Day : Integer range 1 .. 31;
  Month : Month_Name := January;
  Year : Integer range 1 .. 3000 := 2012;
end record;
```

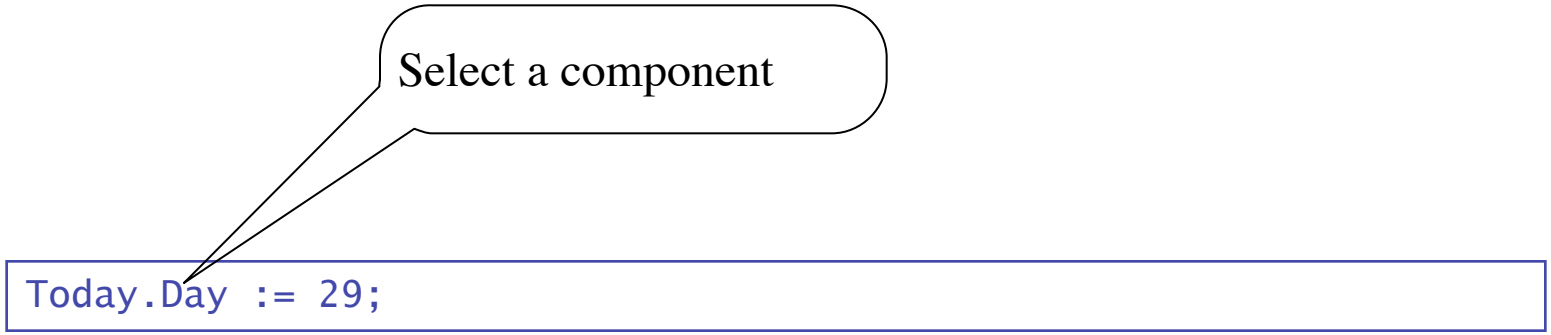
Default value

Records - Literals

```
Today : Date := (31, November, 2012);
Birthday : Date := (Day => 30, Month => February, Year => 2010);
```

By name

Records - Selection



Access type (Pointers)

Declare an access type

```
type Date_Acc is access Date;
D : Date_Acc;
...
D := null;
```

Literal for ‘access to nothing’

Dereferencing

```
type Date_Acc is access Date;
D : Date_Acc;
...
Today : Date := D.all;
J : Day := D.Day
```

Access dereference

Implicit deference for record and array components.
Equivalent to D.all.Day

Allocation (by type)

Allocation (using default values)

```
D : Date_Acc := new Date;
```

Allocation (by type)

Default value is null

Access to unconstrained type

```
type String_Acc is access String;
Msg : String_Acc;
Buffer : String_Acc := new String (1 .. 10);
```

Constraint required

Allocation (by expression)

```
D : Date_Acc := new Date'(30, November, 2011);
Msg : String_Acc := new String'("Hello");
```

Note the tick.
Same notation as type qualification

Mutually recursive types

Incomplete type declaration.
Must be completed in the same declarative region.

```
type Node;
type Node_Acc is access Node;

type Node is record
  Op : Operation;
  Left, Right : Node_Acc;
end record;
```

More about records

Not known at compile time

```
Max_Len : constant Natural := Compute_Max_Len;

type Person is record
  First_Name : String (1 .. Max_Len);
  Last_Name : String (1 .. Max_Len);
end record;
```

Records with discriminant

Discriminant, cannot be modified

```
type Person (Max_Len : Natural) is record
  First_Name : String (1 .. Max_Len);
  Last_Name : String (1 .. Max_Len);
end record;
```

Person is an indefinite type

Records with variant

```
type Node (Op : Operation) is record
  Id : Natural;
  case Op is
     when Unary_Operation =>
        Operand : Node_Acc;
     when Dyadic_Operation =>
        Left, Right : Node_Acc;
  end case;
end record;
```

Variant part. Only one, at the end.

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
Buf : String (1 .. 10);
...
Buf (2 .. 4) := “Ab”;
```

2: Is there a compilation error ?

```
type Person (Max_Len : Natural) is record
  First_Name : String (1 .. Max_Len);
  Last_Name : String (1 .. Max_Len);
end record;

...

A : Person;
```

3: Is there a compilation error ?

```
type Person (Max_Len : Natural) is record
  Name : String (1 .. Max_Len);
end record;

...

A : Person (20);
```

4: Is there a compilation error ?

```
type Person (Max_Len : Natural) is record
  Name : String (1 .. Max_Len);
end record;

...

A : Person := Person'(6, "Pierre");
```

5: Is there a compilation error ?

```
type Person (Max_Len : Natural) is record
  Name : String (1 .. Max_Len);
end record;

...

A : Person := Person'(20, "Pierre");
```

6: Is there a compilation error ?

```
type Date1_Acc is access Date;
type Date2_Acc is access Date;
D1 : Date1_Acc;
D2 : Date2_Acc;
...
D1 := D2;
```

7: Is there a compilation error ?

```
type Date_Acc is access Date;
D1 : Date_Acc := new Date;
D2 : Date_Acc;
...
D1 := D2;
```

8: Is there a compilation error ?

```
type String_Acc is access String;
S : String_Acc := new String'("Hello");
C : Character;
...
C := S.all (0);
```

9: Is there a compilation error ?

```
type String_Acc is access String;
S : String_Acc := new String'("Hello");
C : Character;
...
C := S.all (1);
```

10: Is there a compilation error ?

```
type String_Acc is access String;
S : String_Acc := new String'("Hello");
C : Character;
...
C := S (1);
```

11: Is there a compilation error ?

```
type Node (Op : Operation) is record
  Id : Natural;
  case Op is
     when Unary_Operation =>
        Operand : Node_Acc;
     when Dyadic_Operation =>
        Left, Right : Node_Acc;
  end case;
end record;

N : Node (Op_Plus);
```

12: Is there a compilation error ?

```
type Node (Op : Operation) is record
  Id : Natural;
  case Op is
     when Unary_Operation =>
        Operand : Node_Acc;
     when Dyadic_Operation =>
        Left, Right : Node_Acc;
  end case;
end record;

N : Node := (Op_Unary_Plus, 2, null);
```

13: Is there a compilation error ?

```
type Node (Op : Operation) is record
  Id : Natural;
  case Op is
     when Unary_Operation =>
        Operand : Node_Acc;
     when Dyadic_Operation =>
        Left, Right : Node_Acc;
  end case;
end record;

N : Node (Op_Unary_Plus);
...
... N.Left ...
```

14: Is there a compilation error ?

```
type Node (Op : Operation) is record
  Id : Natural;
  case Op is
     when Unary_Operation =>
        Operand : Node_Acc;
     when Dyadic_Operation =>
        Left, Right : Node_Acc;
  end case;
end record;

N : Node_Acc := ...
...
... N.Left.Op ...
```

AdaCore
The GNAT Pro Company

Privacy

Private part

Not visible from external units

```
package Stacks is
   procedure Hello;

private

   procedure Hello2;
end Stacks;
```

Abstract data types: Declaration

Declare a private type: you cannot depend on its implementation.
You can only assign and test for equality.

```
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);

private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Abstract data types: vocabulary

Partial view

Full view

```
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);

private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Abstract data types

```
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   ...
end Stacks;
```

You shouldn't read the private part to use this package

```
with Stacks; use Stacks;

procedure Test_Stack is
   S : Stack;
   Res : Integer;
begin
   Push (S, 5);
   Push (S, 7);
   Pop (S, Res);
end Test_Stack;
```

Limited types

Limited type.
Cannot assign nor compare

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Pop (S : in out Stack; Val : out Integer);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is limited record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Limited types

Full view is not limited

```
package Stacks is
   type Stack is limited private;
   ...
private
   ...
   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

Full view is limited

```
package Stacks is
   type Stack is limited private;
   ...
private
   ...
   type Stack is limited record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
package Stacks is
   type Stack;

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

private

   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

2: Is there a compilation error ?

```
package Stacks is
   type Stack is private;

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

private

   type Stack is range 1 .. 100;
end Stacks;
```

3: Is there a compilation error ?

```
package Stacks is
   type Stack is private;

   procedure Push (S : in out Stack; Val : Integer);
end Stacks;
```

4: Is there a compilation error ?

```
package Stacks is
   type Stack is private;

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

private

   type Stack is range 1 .. 100;
end Stacks;
```

```
with Stacks; use Stacks;

procedure Test is
  T : Stack;
begin
  T := 3;
end Test;
```

5: Is there a compilation error ?

```
package Stacks is
   type Stack is private;

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

private

   type Stack is range 1 .. 100;
end Stacks;
```

```
with Stacks; use Stacks;

package Stacks2 is
   type Stack2 is record
     S1 : Stack;
     S2 : Stack;
   end record;
end Stacks2;
```

6: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

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

private

   type Stack is range 1 .. 100;
end Stacks;
```

```
with Stacks; use Stacks;

procedure Test is
  T : Stack := 3;
begin
  ...
end Test;
```

7: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   function Init return Stack;
private
   ...
end Stacks;
```

```
with Stacks; use Stacks;

procedure Test is
  T : Stack := Init;
begin
  ...
end Test;
```

8: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   function Init return Stack;
private
   ...
end Stacks;
```

```
with Stacks; use Stacks;

procedure Test is
  T : Stack;
begin
  T := Init;
  ...
end Test;
```

9: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   ...
end Stacks;
```

```
with Stacks; use Stacks;

procedure Test is
  T : Stack;
begin
  Init (T);
  ...
end Test;
```

10: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

```
package body Stacks is
   procedure Init (S : out Stack) is
   begin
     S := (Top => 1, Content => (others => <>));
   end Init;
   ...
end Stacks;
```

11: Is there a compilation error ?

```
package Stacks is
   type Stack is limited private;

   procedure Push (S : in out Stack; Val : Integer);
   procedure Init (S : out Stack);
private
   subtype Stack_Index is Natural range 1 .. 10;
   type Content_Type is array (Stack_Index) of Natural;

   type Stack is limited record
      Top : Stack_Index;
      Content : Content_Type;
   end record;
end Stacks;
```

```
package body Stacks is
   procedure Init (S : out Stack) is
   begin
     S := (Top => 1, Content => (others => <>));
   end Init;
   ...
end Stacks;
```

12: Is there a compilation error ?

```
package P1 is
   type Stack is limited private;

   ...
end P1;
```

```
with P1;
package P2 is
   type T2 is record
     A : P1.Stack;
   end record;
end P2;
```

```
with P2; use P2;

...
  V1, V2 : T2;
...
  V2 := V1;
...
```

AdaCore
The GNAT Pro Company

Generics

Generic declaration

A generic subprogram is not a subprogram!

Formal part

```
generic
   type Elem is private;
procedure Exchange (A, B: in out Elem);
```

```
generic
   type Item is private;
   with function "*" (A, B : Item) return Item is <>;
function Squaring (X : Item) return Item;
```

```
generic
   type Item is private;
package My_Pkg is
  procedure Exchange (A, B: in out Elem);
end My_Pkg;
```

Generic body

```
procedure Exchange (A, B: in out Elem) is
  T : Elem := A;
begin
  A := B;
  B := T;
end Exchange;
```

Generic instantiation

```
procedure Int_Exchange is new Exchange (Integer);
```

Formal type

- **Don't forget that validity of the body is checked during compilation**
  - Not all operators are available with all types
  - A formal type specifies the kind of types
- **Formal types:**
  - **type** T (<>) **is limited private**: any type
  - **type** T **is limited private**: any definite type
  - **type** T (<>) **is private**: any nonlimited type
  - **type** T **is private**: any nonlimited definite type.
  - **type** T **is** (<>): discrete types (enumeration, integer, modular)
  - **type** T **is range** <>: signed integer types
  - **type** T **is mod** <>: modular types
  - **type** T **is digits** <>: floating point
  - **type** T **is delta** <>: fixed point
  - **type** T **is array** ...: array type
  - **type** T **is access** ...: access type
- **Examples:**

```
type Item is private;
type Index is (<>);
type Vector is array (Index range <>) of Item;

type Link is access Item;
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

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

```
procedure P1 is new P (Elem => String);
```

2: Is there a compilation error ?

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

```
procedure P is
  Var : Elem;
begin
  ...
```

AdaCore
The GNAT Pro Company

Exceptions

Exception declaration

```
My_Except : exception;
```

Raising an exception

Execution is abandoned.

```
raise My_Except;
```

Handling an exception

Block (sequence of statements)

Exception to be handled

```
begin
  Open (File, In_File, “input.txt”);
exception
  when E : Name_Error =>
     Put (“Cannot open input file : “);
     Put_Line (Exception_Message (E));
     raise;
end;
```

Reraise current occurrence.

Handling an exception

Block (sequence of statements)

Exception to be handled

```
begin
  Open (File, In_File, “input.txt”);
exception
  when Name_Error =>
     Put (“Cannot open input file“);
end;
```

Handling an exception

Exception occurrence

```
begin
  Open (File, In_File, “input.txt”);
exception
  when E : Name_Error =>
     Put (“Cannot open input file : “);
     Put_Line (Exception_Message (E));
end;
```

Handling an exception

```
begin
  Open (File, In_File, “input.txt”);
exception
  when E : Name_Error =>
     Put (“Cannot open input file : “);
     Put_Line (Exception_Message (E));
     raise;
end;
```

Reraise current occurrence.

Handling an exception

```
   ...
exception
  when Constraint_Error =>
     Put_Line ("Overflow");
     raise;
  when others =>
     Put_Line ("Unexpected exception");
     raise;
end;
```

Predefined exceptions

- **Constraint_Error:**
  - raised when bounds or subtype doesn't match
  - raised in case of overflow (-gnato for GNAT)
  - null dereferenced
  - division by 0

- **Program_Error:**
  - weird stuff (eg: elaboration)

- **Storage_Error:**
  - not enough memory (allocator)
  - not enough stack
    - -fstack-check for GNAT
    - Quizz: implementation ?

- **Tasking_Error**

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
procedure P is
  Ex : exception;
begin
  raise Ex;
end;
```

AdaCore
The GNAT Pro Company

Tasking (83)

Simple Task

Declarations

```
task T;

task body T is
begin
   ...
end;
```

Simple synchronization

```
procedure P is
  task T;

  task body T is
  begin
    for I in 1 .. 10 loop
       Put_Line ("hello");
    end loop;
  end;
begin
   null;
end;
```

Wait until tasks have terminated.

Simple synchronization

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

```
package body P is
  task body T is
  begin
    for I in 1 .. 10 loop
       Put_Line ("hello");
    end loop;
  end;
end;
```

```
with P;

procedure Main is
begin
   null;
end;
```

Wait until tasks have terminated.

Delay

```
task T;

task body T is
begin
  for I in 1 .. 10 loop
     Put_Line ("hello");
     delay 1.0;
  end loop;
end;
```

In seconds

Synchronization: rendez vous

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

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

```
task T1;

task body T1 is
begin
  ...
  T.Start;
  ...
end;
```

Synchronization

Synchronization: rendez vous

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

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

Synchronization: rendez vous

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

A parameter

```
task T1;

task body T1 is
begin
  ...
  T.Start ("Hello");
  ...
end;
```

Synchronization: rendez vous

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

```
task body T is
  accept Start (M : String) do
     Put_Line (M);
  end Start;
end T;
```

```
task T1;

task body T1 is
begin
  ...
  T.Start ("Hello");
  ...
end;
```

Task types

```
task type T is
  entry Start (M : String);
end T;
```

```
task body T is
  accept Start (M : String) do
     Put_Line (M);
  end Start;
end T;
```

```
type T_Acc is access T;

   V1 : T;
   V2 : T_Acc;
begin
   V1.Start ("1");
   V2 := new T;
   V2.all.Start ("2");
...
```

AdaCore
The GNAT Pro Company

Tasking (Ravenscar)

Simple Task

Declarations

```
task T;

task body T is
begin
   ...
end;
```

Cyclic tasks

```
with Ada.Real_Time; use Ada.Real_Time;
…
  task T;

  task body T is
    Next : Time := Clock;
    Cycle : constant Time_Span := Milliseconds (100);
  begin
    delay until Next;
    Next := Next + Cycle;
    ...
  end;
```

Protected objects

Exclusive access to the object.

Operations
(Only subprograms)

```
protected Obj is
  procedure Set (V: Integer);
  function Get return Integer;
private
  Local : Integer;
end Obj;
```

Protected objects: body

```
protected Obj is
  procedure Set (V: Integer);
  function Get return Integer;
private
  Local : Integer;
end Obj;
```

Procedures can modify the data

Functions cannot change the data

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

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

Protected objects: entries

```
protected Obj is
  procedure Set (V: Integer);
  entry Get (V : out Integer);
private
  Value : Integer;
  Is_Set : Boolean := False;
end Obj;
```

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

  entry Get (V : out Integer)
    when Is_Set is
  begin
    V := Local;
    Is_Set := False;
  end Get;
end Obj;
```

Barrier: entry will be blocked until the condition is true.

Protected objects: entries

```
protected Obj is
  procedure Set (V: Integer);
  entry Get (V : out Integer);
private
  Value : Integer;
  Is_Set : Boolean := False;
end Obj;
```

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

  entry Get (V : out Integer)
    when Is_Set is
  begin
    V := Local;
    Is_Set := False;
  end Get;
end Obj;
```

Barrier is evaluated:
- at call
- at exit of procedure or entry

Protected objects: exclusion

Only will task is awoken when the barrier is relieved

```
protected Obj is
  procedure Set (V: Integer);
  entry Get (V : out Integer);
private
  Value : Integer;
  Is_Set : Boolean := False;
end Obj;
```

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

  entry Get (V : out Integer)
    when Is_Set is
  begin
    V := Local;
    Is_Set := False;
  end Get;
end Obj;
```

Protected types

```
protected type Obj is
  procedure Set (V: Integer);
  function Get return Integer;
  entry Get_Non_Zero (V : out Integer);
private
  Local : Integer;
end Obj;
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
task type T;

...

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

2: Is there a compilation error ?

```
task type T;

...

type myrec is record
  N : Natural;
  P : T;
end record;

P1, P2: myrec;

...

  P1 := P2;
```

3: Does this code terminate ?

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

procedure Main is
  Ok : Boolean := False;

  protected O is
    entry P;
  end O;

  protected body O
    entry P when Ok is
    begin
       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;
```

4: Does this code terminate ?

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

procedure Main is
  Ok : Boolean := False;

  protected O is
    entry P;
    procedure P2;
  end O;

  protected body O
    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;
```

5: How does this code terminate ?

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

procedure Main is
  task T is
    entry Start;
  end T;

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

6: When does this procedure terminate ?

```
procedure Main is
  task type T;

  task body T is
  begin
     delay 2.0;
  end T;
  type T_Acc is access T;
  T1 : T_Acc;
begin
  T1 := new T;
end Main;
```

6b: When does this procedure terminate ?

```
task type T;

task body T is
begin
   delay 2.0;
end T;
type T_Acc is access T;

procedure Main is
  T1 : T_Acc;
begin
  T1 := new T;
end Main;
```

7: Is there a compilation error ?

```
procedure Main is
  Ok : Boolean := False;

  protected O is
    function F return Boolean;
  end O;

  protected body O is
    function F return Boolean is
    begin
      Ok := not Ok;
      return Ok;
    end F;
  end O;

  V : Boolean;
begin
  V := O.F;
end;
```

8: Is there a compilation error ?

```
procedure Main is
  protected O is
    function F return Boolean;
  private
    Ok : Boolean := False;
  end O;

  protected body O is
    function F return Boolean is
    begin
      Ok := not Ok;
      return Ok;
    end F;
  end O;

  V : Boolean;
begin
  V := O.F;
end;
```

AdaCore
The GNAT Pro Company

Interfacing

Type convention

```
type T is (E_a, E_b, E_c);

pragma Convention (C, T);
```

Use C representation

Type convention

Provide C type declarations

```
with Interfaces.C; use Interfaces.C;
…
type T is record
  A : int;
  B : long;
  C : unsigned;
end record;

pragma Convention (C, T);
```

Foreign subprograms

```
int my_func (int a);
```

```
with Interfaces.C; use Interfaces.C;
…

function my_func (a : int) return int;
pragma Import (C, my_func);
```

Function imported

Foreign subprograms

```
void My__Func (int a);
```

```
with Interfaces.C; use Interfaces.C;
…

procedure my_func (a : int);
pragma Import (C, my_func, “My__Func”);
```

Procedure imported

Foreign subprograms

```
extern void My__Func (int a);
```

Procedure exported

```
with Interfaces.C; use Interfaces.C;
…

procedure my_func (a : int);
pragma Export (C, my_func, “My__Func”);

procedure my_func (a : int) is
  …
end my_func;
```

Foreign variables

```
extern int my_var;
```

variable exported

```
with Interfaces.C; use Interfaces.C;
…

my_var : int;
pragma Export (C, my_var);
```

Linking with a foreign object file (GNAT)

```
gnatmake main.adb -largs func.o
```

AdaCore
The GNAT Pro Company

Quizz

1: Is there a compilation error ?

```
procedure P;
pragma Import (C, P);

procedure P is
begin
  null;
end;
```

2: Is there a compilation error ?

```
procedure P is
begin
  null;
end;
pragma Export (C, P);
```

3: Is there a compilation error ?

```
procedure P is
  procedure P1;
  pragma Export (C, P1);

  procedure P1 is
  begin
     null;
  end P1;
begin
  null;
end;
```

4: Is there a compilation error ?

```
function Get_Version return String;
pragma Import (C, Get_Version);
```

5: Is there a compilation error ?

```
procedure Put_Str (S : String);
pragma Import (C, Put_Str);
```

AdaCore
The GNAT Pro Company

Low-level

Deallocation

A generic procedure

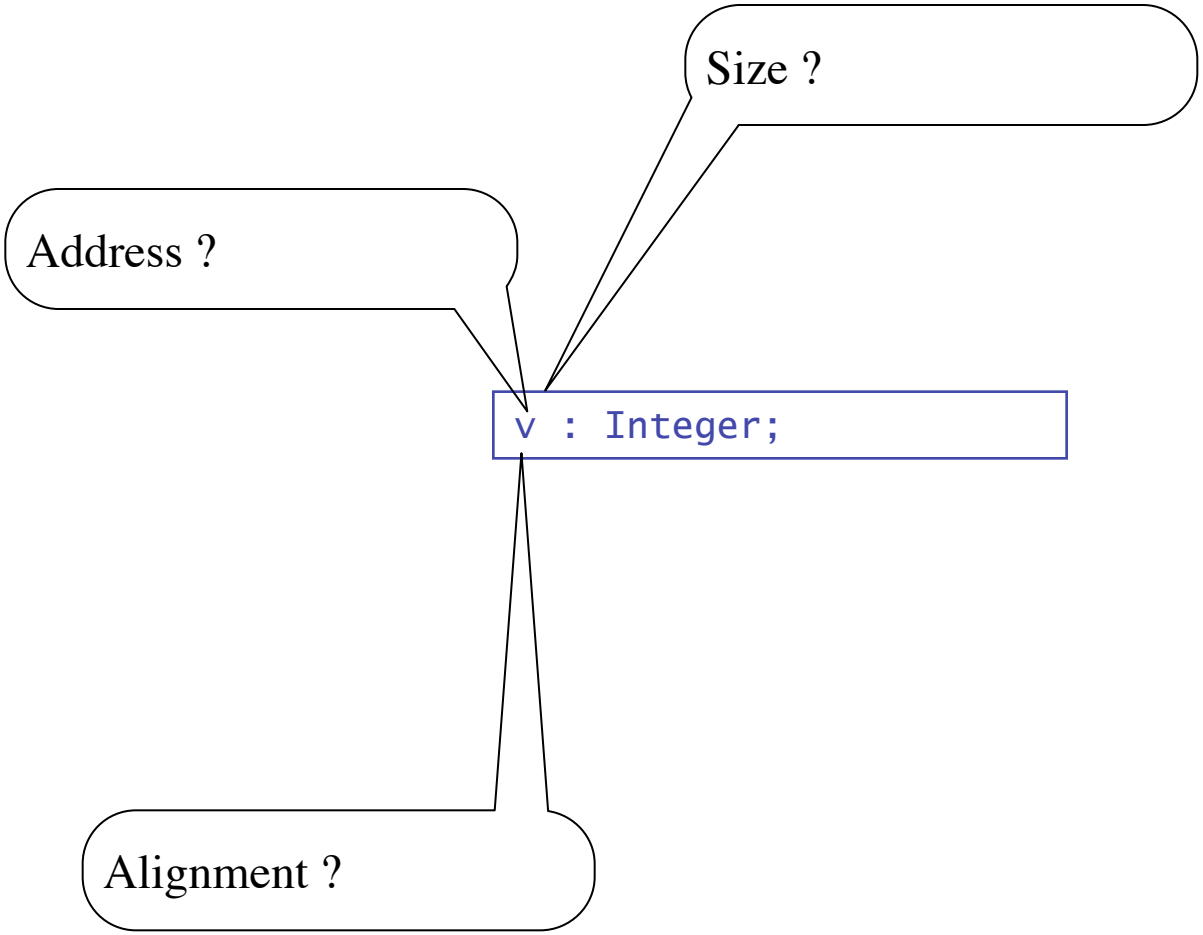
```
with Ada.Unchecked_Deallocation;

…
  type My_Acc is access My_Type;

  procedure Deallocate is new
    Ada.Unchecked_Deallocation (My_Type, My_Acc);
  V : My_Acc;
…
  V := new My_Type;
…
  Deallocate (V);
```

Release the memory and set V to **null.**
(noop if V is already null)

Low level aspects



Querying address

```
with System; use System;

package P is
  V : Integer;

  V_Addr : Address := V'Address;
end P;
```

Address of V

Querying Alignment

```
  V : Integer;
…
     V'Alignment;
```

Alignment for V, in bytes

Querying size

```
  V : Integer;
…
     V'Size;
```

Size for V, in bits

Querying size

Minimal number of bits needed to represent Integer (= 32)

```
Integer'Size
```

```
Natural'Size
```

Minimal number of bits needed to represent Natural (= 31)

Specifying address (1)

Must be correctly aligned

```
with System; use System;

package P is
  V : Integer;
  for V'Address use System.Storage_Elements.To_Address (16#fff_0000#);
end P;
```

Specifying address (2)

GNAT specific attribute

```
with System; use System;

package P is
  V : Integer;
  for V'Address use System'To_Address (16#fff_0000#);
  pragma Import (Ada, V);
end P;
```

Prevent initialisation

Specifying Size

```
with System; use System;

package P is
  V : Integer;
  for V'Size use 32;
end P;
```

Must be large enough

Specifying Alignment

```
with System; use System;

package P is
  V : Integer;
  for V'Alignment use 1;
end P;
```

Can be over or under aligned

Packing Arrays

```
type Bit_Vector is array (0 .. 31) of Boolean;
pragma Pack (Bit_Vector);
```

Size of Bit_Vector would be 32

Packing records

```
type My_Rec is record
  A : Boolean;
  C : Natural;
end record;
pragma Pack (My_Rec);
```

Size of My_Rec would be 32

Specifying record layout

```
type Register is range 0 .. 15;
for Register'Size use 4;

type Opcode is (Load, Inc, Dec, ..., Mov);
for Opcode'Size use 8;

type RR_370_Instruction is record
   Code : Opcode;
   R1   : Register;
   R2   : Register;
end record;

for RR_370_Instruction use record
   Code at 0 range 0 .. 7;
   R1   at 1 range 0 .. 3;
   R2   at 1 range 4 .. 7;
end record;
```

Bit conversion

A generic function

```
with Ada.Unchecked_Conversion;
…
  subtype Str4 is String (1 .. 4);
  function To_Str4 is new
    Ada.Unchecked_Conversion (Integer, Str4);

  V : Integer;
  S : Str4;
…
  S := To_Str4 (V)
```

Bit copy (like memcpy)

Volatile

```
V : Integer;
pragma Volatile (V);
```

Atomic

```
V : Integer;
pragma Atomic (V);
```

Inline

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

The compiler can read the body

AdaCore
The GNAT Pro Company

Statements

Block statement

- **Syntax**

```
[ identifier : ]
[ declare
     declarative_part ]
  begin
    sequence_of_statements
  end [ identifier ];
```

- Can be useful when a declaration is needed in the middle of statements.

AdaCore
The GNAT Pro Company

Expressions and Attributes

Operators

- **Logical operators: and, xor, or**
  - predefined for boolean and modular types
- **Short-circuit operator: and then, or else**
  - Defined only for boolean types
  - RHS is evaluated only if required (cf C && ||)
- **Relational operators: =, /=, <, <=, >, >=**
  - Note that 'not equal' is /=
  - Predefined for scalar types (numerical, enumeration)
- **Binary adding operator: +, -, &**
  - & is the concatenation operator, predefined for one-dimensional array
  - +, - are predefined for numeric types
- **Unary adding operator: +, -**
  - + is identity operator
  - - is negation
  - Predefined for numeric types
- **Multiplying operators: *, /, mod, rem**
  - Predefined for numeric types
  - mod is modular, rem is remainder
    - Slightly different semantic, see ARM 4.5.5

Operators

- **Highest precedence operator: **, abs, not**
  - ** is exponentation
  - ** is predefined for integer types, RHS is natural.
  - ** is predefined for real types, RHS is integer
  - abs is predefined for numeric types
  - not is predefined for boolean and modular types.

- **Membership test**

```
simple_expression [ not ] in range
simple_expression [ not ] in subtype_mark
```

  - True iff the expression is (not) within the range

- **Notation**
  - 'x OP y' can also be written as '"OP" (x, y)'
  - Infix notation vs functional notation

Expressions

- **Grammar**

```
expression ::=
  relation { and relation } | relation { and then relation }
  relation { or relation } | relation { or else relation }
  relation { xor relation }
relation ::=
    simple_expression [ relation_operator simple_expression ]
  | simple_expression [ not ] in range
  | simple_expression [ not ] in subtype_mark
simple_expression ::=
  [ unary_adding_operator ] term { binary_adding_operator term }
term ::=
  factor { multiplying_operator factor }
factor ::=
  primary [ ** primary ] | abs primary | not primary
primary ::=
    numeric_literal | null | string_literal | aggregate
  | name | qualified_expression | allocator | ( expression )
```

- Usual operator priority

Type conversion

- **Convert an expression to a type**

```
type_name ( expression )
```

  - Allowed from numeric type to numeric type
  - Also allowed for some array type
  - Also allowed in other cases
    - More in OOP chapter
  - Breaks the strong-typing rule but still useful
  - Note that 'A (B)' can be:
    - an indexed name
    - a slice name
    - an function call
    - a type conversion
    - an indexed component of an access to an array object
    - an indexed component of the result of a function call
    - ...
  - Hard work for the compiler!

Type qualification

- **Evaluate an expression using a context**

```
type_name‘( expression )
```

  - Used to avoid ambiguity

Attributes

- **An elegant way to retrieve properties**

```
prefix ‘ attribute_designator [ ( static_expression ) ]
```

- **For scalar subtypes:**
  - S’First: lower bound of the range of S
  - S’Last: upper bound
  - S’Range: range
  - S’Image (X): returns an image of the value
  - S’Value (X): returns a value from a string
  - ...
- **For discrete subtypes:**
  - S’Pos (X): Position number of X
  - S’Val (X): Xth value of S
- **For array subtypes and objects:**
  - S’First [(N)]: lower bound of the Nth index
  - S’Last [(N)]: upper bound of the Nth index
  - S’Length [(N)]: length of the Nth index
  - S’Range [(N)]: range of the Nth index

AdaCore
The GNAT Pro Company

Packages

Organization of a program

- **A program is composed of library units, ie**
  - 'top-level' packages
  - 'top-level' subprograms
  - generally one subprogram - the entry point - and many packages.

- **A library unit must be with-ed to be referenced**

```
with library_unit_name {, library_unit_name }
```

  - Example:

```
with Ada.Text_IO;

procedure Greet is
begin
   Ada.Text_IO.Put_Line("Hello, World!");
end Greet;
```

- **Dependencies are therefore explicit**
  - Very different from C/C++
  - A compiler must provide a tool to build correctly a program
    - eg: gnatmake -O greet
- **GNAT choices**
  - One file per library unit
  - .ads for a specification, .adb for a body

Organization of a program (2)

- **Example: logical organization for:**

```
with Ada.Text_IO;

procedure Greet is
begin
   Ada.Text_IO.Put_Line("Hello, World!");
end Greet;
```

```
package Standard is
  ...
  type Boolean is (False, True);
  ...
  type Integer is range ... ;
  function “+” (Left, Right : Integer) return Integer;
  ...

  package Ada.Text_IO is
    ...
    procedure Put_Line (Msg : String);
    ...
  end;

  procedure Greet is
  begin
     Ada.Text_IO.Put_Line("Hello, World!");
  end Greet;
end Standard;
```

AdaCore
The GNAT Pro Company

Predefined Environment

Predefined packages

- **In Ada95, they are children of Ada**
  - Ada.Text_IO: input/output, reading and writing files
  - Ada.Command_Line: program arguments
  - Ada.Containers: linked lists, maps, sets, vectors...
  - Ada.Directories: file system stuff
  - Ada.Numerics: Math stuff
  - Ada.Strings: String handling
  - ...

Ada.Text_IO

- **Basic Output:**
  - Put (Item : Character)
  - Put (Item : String)
    - Write on standard output
  - Put_Line (Item : String)
    - Same as Put but followed by a new line
  - New_Line
    - Write a new line

- **For Integers:**
  - Use Ada.Integer_Text_IO (for Integer type)
  - or instantiate Ada.Text_IO.Integer_IO

Text

- **For Modulars:**
  - Instantiate Ada.Text_IO.Modular_IO

- **For Floats:**
  - Use Ada.Float_Text_IO (for Float type)
  - or instantiate Ada.Text_IO.Float_IO

Ada.Text_IO

- **Basic Input:**
  - Get (Item : out Character)
    - Read from standard input
  - Get_Line (Item : out String; Len : out Natural)
    - Read a line from standard input

- **For other types:**
  - See previous slide

Ada.Text_IO

- **File handling:**
  - Create (File : in out File_Type; Mode : in File_Mode := Out_File; Name : String := “”; Form : String := “”)
    - Create a new file
    - Raise Name_Error if the file already exists
  - Open (File : in out File_Type; Mode : in File_Mode := Out_File; Name : String := “”; Form : String := “”)
    - Open an existing file
    - Raise Name_Error if the file doesn’t exist
  - Close (File : in out File_Type)
    - Close an opened file
  - Delete (File : in out File_Type)
    - Close and delete a file

- **Reading and writing**
  - Use Put, Put_Line, New_Line, Get, Get_Line
    - With File as the first parameter

- **For more details…**
  - See Ada.Text_IO.

Ada.Command_Line

- **Arguments:**
  - function Argument_Count return Natural
    - Number of arguments
  - function Argument (Number : Positive) return String
    - Argument #Number
  - function Command_Name return String
    - Returns the name of the executable

- **Exit status**
  - type Exit_Status is ...
    - type of the exit status
  - Success, Failure
    - Predefined values
  - procedure Set_Exit_Status (Code : Exit_Status)
    - Set the exit status
    - **Doesn't** exit

AdaCore
The GNAT Pro Company

stm32f429 discovery

Core

Core is ARM Cortex M4F

You need a cross-compiler for ARM

GNAT GPL for Bareboard ARM is available on Linux or Windows (works in VM too).

Bareboard

Bareboard environment: no OS

Your code is directly executed, but after runtime initialization.

You need a dedicated runtime - which comes with a very simple graphical library (See screen_interface.ads)

Build

To build the demo:

```
$ gprbuild -P demo.gpr --RTS=./ravenscar-sfp-stm32f4 -XLOADER=RAM
```

To run the demo:

1) Start the debug agent (on Windows: from a CMD console)

```
$ st-util
```

2) Download using gdb

```
$ arm-eabi-gdb obj/hello
(gdb) target remote :4242
(gdb) load
(gdb) c
```

Misc

If you need to build the runtime:

```
$ cd ravenscar-sfp-stm32f4
$ gprbuild
```

Avoid to reflash the board.

The touch screen sensor only has Power-On reset. In some case you need to unplug and replug the board.

Rendu

TP a rendre le 30 janvier

sources de votre projet au format .tar.gz

A envoyer a gingold@adacore.com avec comme sujet:

EPITA 2015 nom_du_groupe

Preciser le nom des membres du groupe.

Bon courage.