

Langages systèmes 1 - Introduction to Ada

Raphael Amiard - Adacore
amiard@adacore.com

Purpose

- Understand what's at stake when choosing a language for a systems project
- Compare two examples of languages designed mainly for systems programming, with a focus on safety/security: Rust & Ada

Evaluation

- 100% TP exercises. No exam, no project.
- However, plan sometime beyond the classes to finish the TPs

Introduction to Ada

- In the seventies, the DOD suffers from an explosion of the number of programming languages used.
- They launch an international competition to design a language that fulfils all the requirements (1974)

- Several propositions
- Winner: Jean Ichbiah's team, CII Honeywell Bull
- First standard of the language language (ANSI, ISO): 1983
- Major revisions in 1995, 2005, 2012.

- General purpose
 - Efficient, simple, implementable
- Safety critical
 - Results in maintainable code
 - Portable
 - Resilient/safe
- Standard should be clear and non ambiguous
- Should work on embedded platforms
- Should handle concurrency/parallelism
- Should allow low-level data handling/hardware interface

- Syntax: Algol/Pascal derivative
- Imperative (like Fortran, Cobol, C/C++, Java, Python...)
- Tasking/parallelism integrated into the language
- Modular (packages, modules, libraries)
- A lot of static AND dynamic checking

Priviledged markets

- Real time systems
- Safety critical/mission critical systems
- Security critical systems

Exemples

- NVIDIA firmware (SPARK)
- Arianne 6
- 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

```
with Ada.Text_IO; use Ada.Text_IO;  
  
-- Display a welcome message  
procedure Greet is  
begin  
    Put_Line ("Hello, world!");  
end Greet;
```

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

Imperative language

```
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;
```

- I here denotes a constant that is only accessible in the loop.
- "1 .. 10" is a range.
- Put_Line is a procedure call. procedure is like a fn returning void in C/C++.

Imperative language - while loops

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
  -- Variable declaration. Only legal in declarative
  -- parts.
  I : Integer := 1;
begin
  -- Condition. *Must* be of type boolean
  while I < 10 loop
    Put_Line("Hello, World!");

    -- Assignment
    I := I + 1;
  end loop;
end Greet;
```

Imperative language - 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;
    -- Exit statement - takes a boolean condition
    I := I + 1;
  end loop;
end Greet;
```

Imperative language - If/Elsif/Else

```
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;
```

Imperative language - If/Else

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
  I : Integer := 1;
begin
  loop
    -- "or else" is the short circuit or operator
    if I = 5 or else I = 12 then
      exit;
    elsif I < 5 and then I > 2 then
      Put_Line ("I = 3 | 4");
    else
      Put_Line("Hello, World!");
    end if;

    I := I + 1;
  end loop;
end Greet;
```

Imperative language - Case statement

```
procedure Greet is
  I : Integer := 0;
begin
  loop
    -- Expression must be of a discrete type. All the
    -- values must be covered.
    case I is
      when 0 => Put_Line ("Starting...");
      when 3 .. 4 => Put_Line ("Hello");
      when 7 | 9 => exit;
      -- 'when others' must be the last one and alone (if
      -- present)
      when others => Put_Line ("Hello, World!");
    end case;
    I := I + 1;
  end loop;
end Greet;
```


Quizz: Imperative language

Quizz 1: Is there a compilation error?

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

Quizz 2: Is there a compilation error?

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

Quizz 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;
```

Quizz 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;
```

Quizz 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;
```

Quiz 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;
```

Quiz 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;
```


Quiz 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;
```

Quiz 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;
```

Quiz 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;
```

Quizz 11: Which one is an error?

```
V      : Integer;  
1V     : Integer;  
V_     : Integer;  
_V     : Integer;  
V__1   : Integer;  
V_1    : Integer;
```

Strongly typed language

What is a type?

- Integer types are just regular types (not built-ins)

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

procedure Greet is
  type My_Int is range 1 .. 20;
  -- Declare a signed integer type, and give the bounds

  -- Like variables, declarations can only happen in
  -- declarative region
begin
  for I in My_Int loop
    Put_Line("Hello, World!");
  end loop;
end Greet;
```

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

procedure Greet is
  type My_Int is range 1 .. 20;
begin
  -- Iterates on every value in the range of My_Int
  for I in My_Int loop
    if I = My_Int'Last then
      --           ^ Higher bound of the type
      -- 'First is the lower bound
      Put_Line ("Bye");
    else
      Put_Line("Hello, World!");
    end if;
  end loop;
end Greet;
```



```
procedure Greet is
  A : Integer := Integer'Last;
  B : Integer;
begin
  B := A + 5;
  -- This operation will overflow, eg. it will
  -- raise an exception at runtime.
end Greet;
```

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

procedure Greet is
  type My_Int is range 1 .. 20;
  A : My_Int := 12;
  B : My_Int := 15;
  M : My_Int := (A + B) / 2;
  -- No overflow here, overflow checks are done at
  -- specific boundaries.
begin
  for I in 1 .. M loop
    Put_Line("Hello, World!");
  end loop;
end Greet;
```

Enumerations

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

procedure Greet is
  type Days is (Monday, Tuesday, Wednesday,
               Thursday, Friday, Saturday, Sunday);
  -- An enumeration type
begin
  for I in Days loop
    case I is
      when Saturday .. Sunday =>
        Put_Line ("Week end!");

        -- Completeness checking on enums
      when others =>
        Put_Line ("Hello on " & Days'Image (I));
        -- 'Image attribute, converts a value to a
        -- String
    end case;
  end loop;
end Greet;
```

```
type Date is record
  -- The following declarations are components of the record
  Day   : Integer;
  Month : Month_Name;
  Year  : Integer;
end record;
```

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

procedure Greet is
  -- Declare two signed types
  type Meters is range 0 .. 10_000;
  type Miles is range 0 .. 5_000;

  Dist_Us : Miles;
  -- Declare a constant
  Dist_Eu : constant Meters := 100;
begin
  -- Not correct: types mismatch
  Dist_Us := Dist_Eu * 1609 / 1000;
  Put_Line (Miles'Image (Dist_Us));
end Greet;
```

```
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);
  --      ^ Type conversion, from Meters to Miles
  -- Now the code is correct

  Put_Line (Miles'Image (Dist_Us));
end;
```

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

procedure Greet is
  C : Character;
  -- ^ Built-in character type (it's an enum)
begin
  C := '?';
  -- ^ Character literal (enumeration literal)

  C := 64;
  -- ^ Invalid: 64 is not an enumeration literal
end Greet;
```

Strong typing

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
  C : Character;
begin
  C := '?';
  Put_Line ("""Ascii"" code of ' & C & ' is"
    --      ^ Use "" to insert quote in a string
    & Integer'Image (Character'Pos (C)));
    --      ^ 'Pos converts a
    --      value to its position

  C := Character'Val (64);
    --      ^ 'Val converts a position to its value
end Greet;
```


Subtypes

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Greet is
  type Days is (Monday, Tuesday, Wednesday, Thursday,
               Friday, Saturday, Sunday);

  -- Declaration of a subtype
  subtype Weekend_Days is Days range Saturday .. Sunday;
  --           ^ Constraint of the subtype
begin
  for I in Days loop
    case I is
      -- Just like a type, a subtype can be used as a
      -- range
      when Weekend_Days =>
        Put_Line ("Week end!");
      when others =>
        Put_Line ("Hello on " & Days'Image (I));
    end case;
  end loop;
end Greet;
```

A subtype doesn't define a type

- All subtypes are of the 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;
  --           ^ Correct: Same type
  Weekend := Monday;
  --           ^ Wrong value for the subtype
  -- Compiles, but exception at runtime
end Greet;
```

Quizz: Types

Quizz 1: Is there a compilation error?

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

Quizz 2: Is there a compilation error?

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

Quizz 3: Is there a compilation error?

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

Quizz 4: Is there a compilation error?

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

Quizz 5: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;  
type My_Int_2 is range Integer'First .. 0;  
type My_Int_3 is range My_Int_2'First .. My_Int_2'Last;
```


Quiz 6: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;  
subtype My_Int_2 is My_Int_1 range 1 .. 100;  
  
V1 : My_Int_1 := 5;  
V2 : My_Int_2;  
V2 := V1;
```

Quizz 7: Is there a compilation error?

```
type My_Int_1 is range 1 .. Integer'Last;  
type My_Int_2 is range 1 .. 100;  
  
V1 : My_Int_1 := 5;  
V2 : My_Int_2;  
V2 := V1;
```

Quizz 8: Is there a compilation error?

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

Quizz 9: Is there a compilation error?

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

Arrays

Array type declaration

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

procedure Greet is
  type My_Int is range 0 .. 1000;

  type My_Int_Array is array (1 .. 5) of My_Int;
  --                               ^ Type of elements
  --                               ^ Bounds of the array
  --                               Can be a type/subtype/anonymous range
  Arr : My_Int_Array := (2, 3, 5, 7, 11);
begin
  for I in Index loop
    Put (My_Int'Image (Arr (I)));
  end loop;
  New_Line;
end Greet;
```

Array index

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

procedure Greet is
  type My_Int is range 0 .. 1000;
  type My_Int_Array is array (11 .. 15) of My_Int;
  --           ^ Low bound can be any value
  Tab : My_Int_Array := (2, 3, 5, 7, 11);
begin
  for I in Index loop
    Put (My_Int'Image (Tab (I)));
    --           ^ Indexation
    --           ^ Will raise an exception when
    --           I = 6
  end loop;
  New_Line;
end Greet;
```

Array index

```
procedure Greet is
  type My_Int is range 1 .. 31;
  type Month is (Jan, Feb, Mar, Apr, May, Jun,
                 Jul, Aug, Sep, Oct, Nov, Dec);

  type My_Int_Array is array (Month) of My_Int;
  --           ^ Can use an enum as the
  --           index

  Tab : constant My_Int_Array :=
  --   ^ constant is like a variable but cannot be
  --   modified
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
  -- Maps months to number of days

  Feb_Days : My_Int := Tab (Feb);
  -- Number of days in February
begin
  for I in Month loop
    Put_Line (My_Int'Image (Tab (I)));
  end loop;
end Greet;
```


Range attribute

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

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

Unconstrained arrays

```
procedure Greet is
  type Int_Array is array (Natural range <>) of Integer;
  -- Indefinite array type
  --                               ^ Bounds are of type Natural,
  --                               but not known

  Numbers : constant Int_Array (1 .. 100) :=
    --                               ^ Specify the bounds
    --                               when declaring
    (0 => 0, others => 100);
  --           ^ Default value
  -- ^ Specify element by index

  Numbers_2 : constant Workload_Type :=
    (1 .. 15 => 8, others => 7);
  -- ^ Range association
begin
  for I in Numbers'Range loop
    Put_Line (Integer'Image (Numbers (I)));
  end loop;
end Greet;
```

Predefined array type: String

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

procedure Greet is
  Message : String (1 .. 11) := "dlrow olleH";
  --      ^ Pre-defined array type.
  --      Component type is Character
begin
  for I in reverse 1 .. 11 loop
    ^ Iterate in reverse order
    Put (Message (I));
  end loop;
  New_Line;
end Greet;
```

Predefined array type: String

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

procedure Greet is
  Message : constant String := "Hello World";
  --           ^ Bounds are automatically computed
  --           from initialization value
begin
  for I in reverse Message'First .. Message'Last loop
    --           ^ 'First and 'Last return the low and
    --           high bound
    -- (But you should use 'Range most of the time)
    Put (Message (I));
  end loop;
  New_Line;
end Greet;
```

Declaring arrays

```
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");
  -- Initial value given by aggregate
begin
  for I in Names'Range loop
    Put_Line (Names (I));
  end loop;
end Greet;
```

Declaring arrays

```
-- Those two declarations produce the same thing
A : String := "Hello World";
B : String (1 .. 11) := ('H', 'e', 'l', 'l', 'o', ' ',
                        'W', 'o', 'r', 'l', 'd');
```

Quizz: Arrays

Quizz 1: Is there a compilation error ?

```
-- Natural is a pre-defined subtype.  
subtype Natural is Integer range 0 .. Integer'Last  
  
type Arr is array (Natural) of Integer;  
Name : Arr;
```


Quizz 2: Is there a compilation error ?

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

Quizz 3: Is there a compilation error ?

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

Quizz 4: Is there a compilation error ?

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

Quizz 5: Is there a compilation error ?

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

Quizz 6: Is there a compilation error ?

```
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;
```

Modular/Structured programming

```
package Week is  
end Week;
```

Packages:

- Group related declarations together
- Define an interface (API)
- Hide the implementation
- Provide a name space

```
package Week is

  -- This is a declarative part. You can put only
  -- declarations here, no statements

  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;
```

Different from header files in C/C++ because:

- Language level mechanism (not a preprocessor)
- Not text based
- With'ing a package does not “copy/paste” the content of the spec into your file
- With GNAT, packages specs go in .ads files (here, it would be week.ads)

With-ing a package

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

with Week;
-- references the package, and adds a dependency on
-- that package

procedure Greet is
begin
  for I in Week.Workload'Range loop
    --      ^ We reference items of the package by prefixing
    --      by package name
    Put_Line (Integer'Image (Workload (I)));
  end loop;
end Greet;
```

Using a package

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

with Week;
use Week;
-- Brings the content of the package in the current
-- namespace

procedure Greet is
begin
  for I in Workload'Range loop
    --      ^ We can reference items of the package directly
    --      now
    Put_Line (Integer'Image (Workload (I)));
  end loop;
end Greet;
```

Package body

```
package body Week is

  -- The body contains additional declarations, not
  -- visible from the spec, or anywhere outside of the
  -- body
  type WorkLoad_Type is array (Days range <>) of Natural;
  Workload : constant Workload_Type :=
    (Monday .. Friday => 8, Friday => 7, Saturday | Sunday => 0);

  function Get_Workload (Day : Days) return Natural is
  begin
    return Workload (Day);
  end;
  -- This is a function. It has a return statement
  -- (mandatory for functions)
end Week;
```

- With GNAT, packages bodies go in .adb files (here, it would be week.adb)

Subprograms

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

-- Here we declare and define a procedure without
-- parameters
procedure Greet is
begin
    Put_Line("Hello, World!");
end Greet;
```

```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);

  function Get_Workload (Day : Days) return Natural;
  -- We declare (but don't define) a function with one
  -- parameter, returning a Natural integer
end Week;
```

```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);

  function Get_Day_Name
    (Day : Days := Monday) return String;
    --           ^ We can return any type,
    --           even indefinite ones
    --           ^ Default value for parameter
end Week;
```

```
package body Week is
  -- Implementation of the Get_Day_Name function
  function Get_Day_Name (Day : Days := Monday) return String is
  begin
    case Day is
      when Monday => return "Monday";
      when Tuesday => return "Tuesday";
      ...
      when Sunday => return "Sunday";
    end case;
  end Get_Day_Name;
end Week;
```



```
package Week is
  type Days is (Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday, Sunday);
  procedure Next_Day (Res : out Days;
    --           ^ Mode out: not initialized at
    --           the beginning, procedure body can
    --           assign it. (Like a return value)
    Day : Days);
    --           ^ No mode: defaults to "in",
    --           which is like a constant
end Week;
```

```
--          You can declare several params at the same
--          v time
procedure Swap (A, B : in out Integer)
--          ^ In out is initialized at the
--          beginning with value passed by
--          caller. procedure can assign it,
--          caller will get new value. Like
--          pass by reference
--          Use it instead of a pointer in C
is
    Tmp : Integer;
begin
    Tmp := A;
    A := B;
    B := Tmp;
    return;
    -- Return statement. Optional for procedures
end Swap;
```

Subprogram call

```
procedure Test_Swap
is
    X, Y : Integer;
begin
    X := 5;
    Y := 7;
    Swap (X, Y);
    --      ^ Positional parameters
    Swap (A => X, B => Y);
    --      ^ Named parameters
    Swap (B => X, A => Y);
    --      ^ You can reverse the order
end Test_Swap;
```

Function calls

```
function Double (I : Integer) return Integer is
begin
    return I * 2;
end Double;

function Quadruple (I : Integer) return Integer is
    Res : Integer := Double (Double (I));
    --           ^ Calling the double function
begin
    Double (I);
    -- ILLEGAL: You cannot ignore a function's return value

    return Res;
end Quadruple;
```

```
function Pi return Float is
--      ^ No parameters, no parentheses
begin
    return 3.1419;
end Pi;

A : Float := Pi;
--      ^ No parameters, no parentheses at call site
--      either!
```

Mutually recursive subprograms

```
procedure Compute_A (V : Natural);  
-- Forward declaration of Compute_A  
  
procedure Compute_B (V : Natural) is  
begin  
    if V > 5 then  
        Compute_A (V - 1);  
        -- ^ Call to Compute_A  
    end if;  
end Compute_B;  
  
procedure Compute_A (V : Natural) is  
begin  
    if V > 2 then  
        Compute_B (V - 1);  
        -- ^ Call to Compute_B  
    end if;  
end Compute_A;
```

Nested subprograms

```
function Quadruple (I : Integer) return Integer is

    function Double (I : Integer) return Integer is
    begin
        return I * 2;
    end Double;
    -- Nested function

begin
    return Double (Double (I));
end Quadruple;
```

Quizz: Packages & subprograms

Quizz 1: Is there a compilation error ?

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

Quizz 2: Is there a compilation error ?

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

Quizz 3: Is there a compilation error ?

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

Quizz 4: Is there a compilation error ?

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

Quizz 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;
```

Quizz 6: Is there a compilation error ?

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

Quizz 7: Is there a compilation error ?

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

Quizz 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;
```


Quiz 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;
```

Quizz 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;
```

Quiz 11: Is there a compilation error ?

```
-- pkg1.ads
package Pkg1 is
    procedure Proc;
end Pkg1;

-- pkg2.ads
with Pkg1;
package Pkg2 is
    ...
end Pkg2;

-- main.adb
with Pkg2;

procedure Main is
begin
    Pkg1.Proc
end Main;
```

Quiz 12: Is there a compilation error ?

```
-- pkg1.ads
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;

-- pkg2.ads
with Pkg1; use Pkg1;
package Pkg2 is
  ...
end Pkg2;

-- pkg2.adb
package body Pkg2 is
  procedure Foo is
  begin
    Proc;
  end Foo;
end Pkg2;
```

Quiz 13: Is there a compilation error ?

```
-- pkg1.ads
package Pkg1 is
procedure Proc;
...
end Pkg1;

-- pkg2.ads
with Pkg1; use Pkg1;
package Pkg2 is
...
end Pkg2;

-- pkg2.adb
with Pkg1; use Pkg1;
package body Pkg2 is
...
end Pkg2;
```

Quiz 14: Is there a compilation error ?

```
-- pkg1.ads
package Pkg1 is
  procedure Proc;
  ...
end Pkg1;

-- pkg2.ads
with Pkg1;
package Pkg2 is
  ...
end Pkg2;

-- pkg2.adb
use Pkg1;

package body Pkg2 is
  procedure Foo is
  begin
    Proc;
  end Foo;
end Pkg2;
```

Privacy

```
package Stacks is
  procedure Hello;

private

  procedure Hello2;
  -- Not visible from external units
end Stacks;
```


Abstract data types: Declaration

```
package Stacks is
  type Stack is private;
  -- Declare a private type: You cannot depend on its
  -- implementation. You can only assign and test for
  -- equality.

  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

```
package Stacks is
  type Stack is private;
  -- Partial view

  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;
  -- Full view
  end record;
end Stacks;
```

Abstract data types

```
-- No need to read the private part to use the package
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;
```

```
-- Example of use
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;
```

Quizz: Privacy

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;
```

Quizz 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;
```

Quizz 3: Is there a compilation error?

```
package Stacks is
  type Stack is private;
  procedure Push (S : in out Stack; Val : Integer);
end Stacks;
```

Quiz 4: Is there a compilation error?

```
-- stacks.ads
package Stacks is
  type Stack is private;
  procedure Push (S : in out Stack; Val : Integer);
  private
    type Stack is range 1 .. 100;
end Stacks;

-- test.adb
with Stacks; use Stacks;
procedure Test is
  T : Stack;
begin
  T := 3;
end Test;
```


Quiz 5: Is there a compilation error?

```
-- stacks.ads
package Stacks is
  type Stack is private;
  procedure Push (S : in out Stack; Val : Integer);
private
  type Stack is range 1 .. 100;
end Stacks;

-- stacks2.ads
with Stacks; use Stacks;

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

Pointers and dynamic allocation

- As much as possible, pointer usage is discouraged in Ada
- Thanks to flexible parameter passing & unconstrained types, you can go pretty far without pointers & dynamic allocation.
- Most of safety critical/embedded apps don't use them
- However, not all applications can avoid pointers
- When they're unavoidable, Ada tries to make them as safe as possible (pre linear typing)

```
procedure Test_Accesses is
  type Integer_Access is access Integer;

  I : Integer := 12;

  IA : Integer_Access := new Integer;
begin
  I.all := 12;

  IA := I'Access;
  --    ^ ILLEGAL: You can't take an access on the stack
  --      by default

  -- NO DEALLOCATION BY DEFAULT
end;
```

```
procedure Test_Accesses is
  type Integer_Access is access all Integer;
  -- ^

  I : aliased Integer := 12;
  IA : Integer_Access := I'Access;
begin
  null;
end;
```

```
procedure Test_Accesses is
  type Integer_Access is access all Integer;

  procedure Inner is
    I  : aliased Integer := 12;
    IA : Integer_Access := I'Access;
    -- ILLEGAL: I is defined one level below Integer_Access

    type Integer_Access_2 is access all Integer;
    IA2 : Integer_Access_2 := I'Access;
    -- LEGAL

  begin
    null;
  end;
begin
  null;
end;
```

```
with Ada.Unchecked_Deallocation;

procedure Test_Accesses is
  type Integer_Access is access Integer;
  IA : Integer_Access := new Integer'(12);

  procedure Free is new Ada.Unchecked_Deallocation
    (Integer, Integer_Access);
begin
  Free (IA);
  -- IA is set to null afterwards
end;
```

- Accessibility level is broken, but was meant to solve the same problems as linear types in Rust.
- No built-in pointer arithmetic: vastly not necessary. Special packages when necessary.
- Special conveniences for unchecked conversions and stuff like that.

More about records

More about records

```
Max_Len : constant Natural := Compute_Max_Len;  
--           ^ Not known at compile time  
  
type Person is record  
  First_Name : String (1 .. Max_Len);  
  Last_Name  : String (1 .. Max_Len);  
end record;
```

```
type Person (Max_Len : Natural) is record
  --      ^ Discriminant. Cannot be modified once initialized.
  First_Name : String (1 .. Max_Len);
  Last_Name  : String (1 .. Max_Len);
end record;
-- Person is an indefinite type (like an array)
```

Records with variant

```
type Node_Acc is access Node;

type Op_Kind is (Bin_Op, Un_Op);
-- A regular enum

type Node (Op : Op_Kind) is record
--      ^ The discriminant is an enum
  Id : Natural;
  case Op is
    when Un_Op =>
      Operand : Node_Acc;
    when Bin_Op =>
      Left, Right : Node_Acc;
      -- Those fields only exist when Op is Bin_Op
  end case;
  -- Variant part. Only one, at the end of the record
  -- definition
end record;
```

Safety in Ada

- The language is safe by default
- No “unsafe” subset, but rather unsafe operations (Unchecked_Deallocation, Unchecked_Conversion, ...)
- Safe means:
 - Very little undefined behavior. Most of Ada’s behavior is defined (check the Ada reference manual if you don’t believe me <https://ada-lang.io/docs/arm>)
 - No memory corruption
 - Illegal operations will result in a runtime exception
- But it also means:
 - Make it easy to write readable, maintainable programs
 - Make it easy to **specify** as much about the program as you can.
 - The behavior cannot always be verified at compile time, but it will be verified at runtime/can be verified by static-analyzers/provers.