The Ada Programming Language

ELTE, IK, PNYF
Zsók Viktória, Ph.D.
zsv@elte.hu

A good programming language

- Is abstract enough
- Has good syntax and it is easy to follow
- Provides good programming tools
- Has dedicated tools to handle the complexity of programs
- Has easy, understandable semantics

Why Ada?

Pros:

- Clear, logical structure
- Differs from other imperative languages
- Contains the necessary properties to introduce a Pascal-like language for students

Cons:

- It can be difficult for beginners
- Less groups are using it

History

- Originally sponsored by US defense United States Department of Defense) organizes the development of HOL (High Order Language)
- Design a language that defines requirements, is competitive and parallel

Requirements

- Establishing requirements
- evaluating the existing languages:
- FORTRAN, COBOL, PL/I, HAL/S,
- TACPOL, CMS-2, CS-4, SPL/1, J3B, Algol 60,
- Algol 68, CORAL 66, Pascal, SIMULA 67,
- LIS, LTR, RTL/2, EUCLID, PDL2, PEARL,
- MORAL, EL-1

Augusta Ada Byron

- The need for new language: none of the existing ones fulfilled the requests
- Pascal, PL/I, ALGOL 68 used as starting point
- Named after Augusta Ada Byron (1815–1852), Countess of Lovelace, daughter of the poet Lord Byron
- Assistant of Charles Babbage (1791 –1871), an English polymath. He was a mathematician, philosopher, inventor and mechanical engineer, who is best remembered now for originating the concept of a programmable computer
- She was the first programmer, worked on mechanical analytical machine

Some of the main features

- An extremely strong, static and safe type system, which allows the programmer to construct powerful abstractions
- Modularity
- Information hiding: the language separates interfaces from implementation
- Portability
- Standardisation: Ada compilers all support exactly the same language; the only dialect, SPARK

Features

- Ada was originally targeted at embedded and real-time system
- strong typing, modularity mechanisms (packages), run-time checking, parallel processing (tasks, synchronous message passing, protected objects, and nondeterministic select statements), exception handling, and generics.

Versions

- Ada 83 based on Pascal
- Feature from: Euclid, Lis, Mesa, Modula,
- Sue, Algol 68, Simula 67, Alphard, CLU
- Ada 95 included: interfaces, parallel programming features, oo classes
- Ada 2005
- Ada 2012

References

- Ada Reference Manual - Language and Standard Libraries
- The Rationale for Ada
- John Barnes: Programming in Ada 2005
- http://en.wikibooks.org/wiki/Ada_Programming
- http://www.ada2012.org/
- And many more books and web-pages

Course requirement

- 2 programming assessments (mandatory homeworks)
- 2 lab exams – 2 marks
- 1 theoretical test – 1 mark
- Final mark: average of the 3 above marks

Ada structure

- Subprograms: procedures and functions
- Hierarchically included
- Packages with interfaces and implementations
- Generics
- Tasks
- Protected objects

The structure of a program

- Including parts of the standard libraries
- Specification: variables, types
- Implementation: statements and maybe an exception handling part

Getting started

```
with Ada.Text_IO;

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

Hello, world! - 2

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

procedure Hello is

begin
   Put_Line("Hello, world!");
   New_Line;
   Put_Line("I am an Ada program with package use.");
end Hello;
```

Compiling, running

- gnatmake hello.adb
- ./hello
- GPS – GNAT Programming Studio
- Emacs Ada-mode

Declaration part

- Apart from statements
- Variables, constants, exceptions, program unit declarations

N: Natural;

- Initial value;

B: Boolean := True;

- More variables of the same type / value

I, J: Integer;

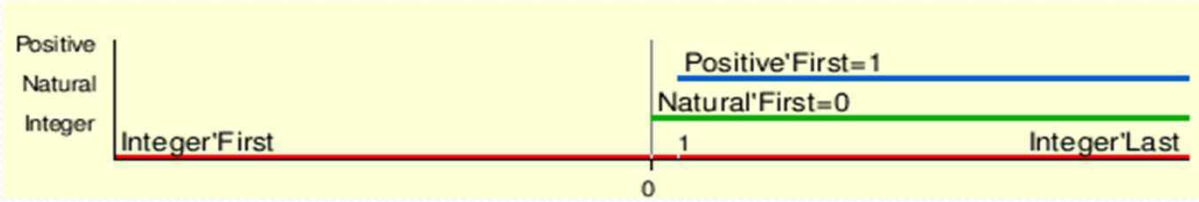
A, B: Positive := 3;

Max: constant Integer := 100;

- Integer, Natural, Positive, Boolean, Character, Float, String

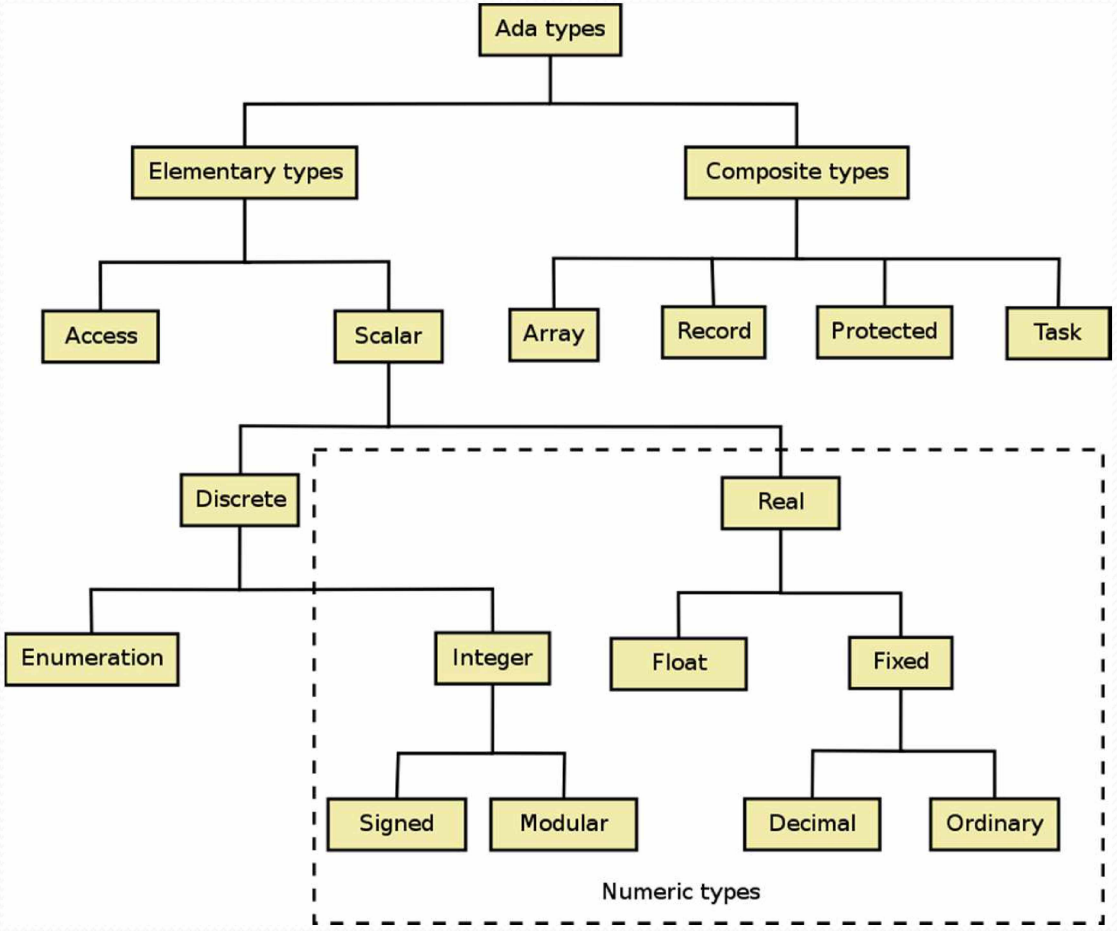
Predefined types

- Integer a value in (-2)**15+1 .. +2**15-1 interval, defines Natural and Positive as subtypes



- Float (define your own floating-point types, and specify your precision and range)
- Duration – a period of time in seconds
- Character - a special form of enumerations
- String – an array of characters
- Boolean - an enumeration of False and True

Type hierarchy



Type and subtype

- There is an important distinction between **type** and **subtype**: a type is given by a set of values and their operations.
- A subtype is given by a type, and a *constraint* that limits the set of values.
- Values are always of a type.
- Objects (constants and variables) are of a subtype

Boolean type

- Included in the Standard package (automatically can be used)
- Needed by *if* and *while*
- Comparison of elements = /= < > <= >=
- predefined operators:
- *Not and or xor and then or else*

if B = True then if B then

if B = False then if not B then

Logical operators, short circuit control forms

- Lazy evaluation: and then, or else
- ***If the value can be determined from the first argument then the evaluation stops***

```
while (I <= N) and then (i mod 2 = 0) loop
do_something;
end loop;
```

```
if A>B and then F(A,B) then
```

- Eager evaluation: and, or
- ***Evaluates both operators***

Integer

- The set of integer values:
- **…, -3, -2, -1, 0, 1, 2, 3, …**
- Predefined operators
- ***+A -A A+B A-B A*B A/B***
- ***A rem B, A mod B, abs A, A**B***
- Integer division truncates towards zero
- Exponentiation raises the first to the power of second (must be integer, positive integer or float, integer)

Differences between rem and mod

- **A B A/B A rem B A mod B**
- **12 5 2 2 2**
- **-12 5 -2 -2 3**
- **12 -5 -2 2 -3**
- **-12 -5 2 -2 -2**
- A=(A/B)*B +(A rem B) takes the sign of A
- A=B*N +(A mod B) takes the sign of B (N is integer)

Float

- *Predefined operators*
- *+X -X X+Y X-Y X*Y X/Y X**Y*
- Exponentiation: Y integer

Mixed values

- Not allowed:

I: Integer := 3;

F: Float := 3.14;

I := F + 1; F := I - 1.3; -- error

- Explicit conversion is needed:

I := Integer(F) + 1; F := Float(I) - 1.3;

Integer(F) will be truncated 1.4 – 1, 1.6 -- 2

1.5 – 2, -1.5 -- 2

Assignment

- Assigns an expression to a variable

I := 5;

- The type of the value and variable must be the same

I := True; -- error

- Subtypes are checked in runtime
- The assigment is not an operator an d it is not an expression (can not be overloaded)
- There is no simultaneous assignment

The empty statement

```
procedure Nothing is
begin
null;
end;
```

- Empty begin-end can not be written
- Used in tasks
- The keyword is multiply used

Control structures

```
if Boolean expression
then
   statements
elsif Boolean expression
then
   statements
else
   statements
end if;
```

Swap

```
if A>B then
Temp := A;
A := B;
B := Temp;
end if;

if A>B then
A := A - B;
else
B := B - A;
end if;
```

Undefined else

```
if (a>0)
    if (b>0)
        c = 1;
          else
c = 0;
```

- Not well defined in: C++, Java, Pascal etc.
- In Ada an if is closed, so we know the where is the else included

Example for if

```
with Ada.Text_IO; use  Ada.Text_IO; …
type Degrees is new Float range -273.15 .. Float'Last;
Temperature : Degrees;
…
if Temperature >= 40.0 then
    Put_Line ("It's extremely hot");
elsif Temperature >= 20.0 then
    Put_Line ("It's warm");
elsif Temperature >= 0.0 then
    Put_Line ("It's cold");
else
    Put_Line ("It's freezing");
end if;
```

Case structure

```
case  X  Is
   when 1 =>
      Walk_The_Dog;
   when 5 =>
      Eat_The_Lunch;
   when 6 | 10 =>
      Sell_All_the_Products;
   when others =>
      Do_as_usuall;
end case;
-- The subtype of X must be a discrete type, i.e. an
enumeration or integer type.
```

Case example

```
case (X mod 20) + 1 is
when 1..3 | 5 | 7 | 11 | 13 | 17 | 19 =>
    F := True;
when others =>
    F := False;
end case;
```

Loops

```
while Boolean expression loop
   statements;
end loop;
```

Example:

```
 while X <= 5 loop
      X := Calculate_Something;
 end loop;
```

example

```
with Ada.Integer_Text_IO;
procedure ten is
I: Positive := 1;
begin
while I <=10 loop
    Ada.Integer_Text_IO.Put( I );
    I := I + 1;
end loop;
end ten;
```

Loops

```
for variable in range loop
   statements;
end loop;
```

```
s := 0;
for i in 1..10 loop
     s := s + i;
end loop;
```

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

Example

```
with Ada.Integer_Text_IO;
procedure ten is
begin
for I in 1..10 loop
Ada.Integer_Text_IO.Put( I );
end loop;
end ten;
```

- The step can be only one,
- I is local variable an should not be declared, can not be changed in for

Reverse example

```
for I in reverse 1..10 loop
      Ada.Integer_Text_IO.Put( I );
end loop;
```

Endless loop

```
loop
      Do_Something;
end loop;
```

Example:

```
 loop
      X := Calculate_Something;
      exit when X > 10;
   end loop;
```

Example

```
loop
Get(Ch);
...
exit when Ch = ‘q’;
...
end loop
```

Programming in Ada course

Review
Block structure
Functions, procedures
Packages
Examples

The block structure

```
declare
   some declarations;
begin
   statements;
exception
   handlers;
end;
```

Hierarchy

```
…
declare
…
begin
…
      declare …
            begin …
            end;
…
end;
```

Module structures

```
procedure P(parameters: in out type) is
   declarations;
begin
   statements;
exception
   handlers;
end P;
```

Functions

```
function F (parameters : in TypeIn) return TypeOut is
   declarations;
begin
   statements;
exception
   handlers;
end F;
```

Subprograms and return

- We call the subprograms by their name

Ada.Text_IO.Put_Line(“Some text");

Ada.Text_IO.Get(Ch);

- Ch a variable of type Character

Text_IO.New_Line;

- After return the subprogram is ended
- At functions, we must write the result after a return statement !

return X+Y;

Function

```
function Factorial ( N: Natural ) return Positive is
Fact: Positive := 1;
   begin
      for I in 1..N loop
          Fact := Fact * I;
      end loop;
   return Fact;
end Factorial;
```

Function – GCD example

```
function GCD ( A, B : Positive ) return Positive is
    X: Positive := A;
    Y: Natural := B;
    Tmp: Natural;
begin
    while Y /= 0 loop
        Tmp := X mod Y;
        X := Y;
        Y := Tmp;
     end loop;
   return X;
end GCD;
```

Procedure

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

Parameters

- **in** : transfers information from the caller

Caller -> called

- **out** : transfers the information computed in a procedure to the caller

Called -> caller

- **in out** :
- – transfers information to the procedure from the caller
- – transfers information from the procedure to the caller

Caller <--> called

Default is **in** !!!

Out or **in out** can be a left value

Example

```
procedure Ex ( Vi: in Integer; Vo: out Integer;
                        Vio: in out Integer) is
begin
   Vio := Vi + Vo; -- error Vo can not be read
   Vi := Vio; -- error, Vi can not be written
   Vo := Vi; -- good
   Vo := 2*Vo+Vio; -- good, Vo already has a value
end Ex;
```

Examples

- procedure Put (Item: in Integer);
- procedure GCD (A, B: in Positive; gd: out Positive);
- procedure Swap (A, B: in out Integer);

- **in** can be only read, we can not assign values
- **out** parameter can be written
- **in out** parameter can be read and written

Example

```
procedure eucl( A, B: in Positive; GCD, LCM: out Positive ) is
    X: Positive := A;
    Y: Positive := B;
begin
    while X /= Y loop
        if X > Y then X := X - Y; else Y := Y - X; end if;
    end loop;
    GCD := X;
    LCM := A * B / GCD ;
end eucl;
```

Recursive function

```
function Factorial ( N: Natural ) return Positive is
begin
    if N > 1 then return N * Factorial (N-1);
                else return 1;
    end if;
end Factorial;
```

Packages

- Logical entities that are connected
- Defines the set of variables, types
- More complex then a subprogram (function or procedure)
- Subprograms are executed
- Packages have components that can be used

The structure of a package

```
package A is
...
end A;
package body A is
...
end A;
```

In a demo program has to be imported:

```
with A; use A;
```

Package specification

- Can not contain implementation details
- Contains declarations of subprograms together with types and variables
- .ads

```
package Ada.Text_IO is
...
procedure Put_Line( Item: in String ) ;
...
end Ada.Text_IO;
```

Package implementation

- Contains the bodies of the subprograms
- .adb

```
package body Ada.Text_IO is
...
procedure Put_Line( Item: in String ) is
...
end Put_Line;
...
end Ada.Text_IO;
```

Compiling

- The specification and the implementation can be compiled separately (2 units)
- A subprogram is only one compilation unit

- Why to write packages?
- Specifying interfaces
- Encouraging team work
- Parallel software development

Packages in other languages

- C++: class, namespace
- Java: class/interface, package
- Modula-2, Clean: module

- The use statement: includes the predefined package
- It can be applied only with packages

- C++: using namespace
- Java: import

```
package math is
   function gcd ( A, B : Positive ) return Positive;
   function factorial( N: Natural ) return Positive
end math ;  -- this is the math.ads file

package math is
   function gcd ( A, B : Positive ) return Positive is
   begin
    ...
    end gcd;

   function factorial( N: Natural ) return Positive is
   begin
   ....
   end factorial;
end math ;  -- this is the math.adb file
-- in a main.adb program the package has to be imported:
-- with math; use math;
```

SubTypes and Derived Types
Arrays, Records

Strongly typed

- Ada is strongly typed - different types of the same family are incompatible with each other;
- a value of one type cannot be assigned to a variable from the other type
- "mixed mode" expression trigger a compilation error
- type conversions must be made explicit

```
type Meters is new Float;
type Miles is new Float;
Dist_Imperial : Miles;
Dist_Metric : constant Meters := 100.0;
 Dist_Imperial := (Miles (Dist_Metric) * 1609.0) / 1000.0; --
Type conversion, from Meters to Miles
```

Type conversion

```
 function To_Miles (M : Meters) return Miles is  begin
return (Miles (M) * 1609.0) / 1000.0;   end To_Miles;

Dist_Imperial : Miles;

Dist_Metric   : constant Meters := 100.0;begin
Dist_Imperial := To_Miles (Dist_Metric);

Put_Line (Miles'Image (Dist_Imperial));
```

Derived types

- you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing

type Social_Security_Number is new Integer range 0 .. 999_99_9999;

- Social_Security_Number *derived type*; its *parent type* is Integer
- you can refine the valid range when defining a derived scalar type (such as integer, floating-point and enumeration)

Subtypes

- types may be used in Ada to enforce constraints on the valid range of values
- enforce constraints on some values while staying within a single type
- a subtype does not introduce a new type
- subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
subtype Degree_Celsius is Float;
subtype Liquid_Water_Temperature is Degree_Celsius range 0.0 .. 100.0;
subtype Running_Water_Temperature is Liquid_Water_Temperature;
```

Subtypes as type aliases

- type aliases generate alternative names — *aliases* — for known types
- type aliases are sometimes called *type synonyms*
- using subtypes without new constraints

```
subtype Meters is Float;
subtype Miles is Float;
Dist_Imperial : Miles;
```

- we don't get all of the benefits of Ada's strong type checking
- recommendation is to use strong typing

```
Type X is new Y;
```

Array type declaration

- composite type / same typed elements
- define contiguous collections of elements that can be selected by indexing
- type My_Int is range 0 .. 1000;
- type Index is range 1 .. 5;
- type My_Int_Array is array (Index) of My_Int;

```
        --                                  ^ Type of elements
Arr : My_Int_Array := (2, 3, 5, 7, 11);
 --                 ^ Array literal, ^ aggregate in Ada
```

Arrays

- specify the index type for the array
- any discrete type is permitted to index an array, including Enum types
- querying an element of the array at a given index, same syntax as for function calls: array object followed by the index in parentheses
- initialize by aggregate
- bounds of an array can be any values
- bounds can vary, you should not assume / hard-code specific bounds when iterating / using arrays

Arrays

```
type Month_Duration 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 Month_Duration;
 --                                  ^ Can use an enumeration type as index
Tab : constant My_Int_Array := (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
--  Maps months to number of days (ignoring leap years)
--  constant declarations cannot be modified
Feb_Days : Month_Duration := Tab (Feb);   --  Nr of days in February
for M in Month loop
 Put_Line (Month'Image (M) & " has " & Month_Duration'Image
                                                (Tab (M))  & " days.");
```

Arrays - Indexing

- indexing operation is strongly typed
- wrong type to index the array - a compile-time error.
- arrays in Ada are bounds checked
- outside of the bounds - run-time error, no random access
- explicitly created an index type for the array or a range of values

```
type My_Array is array (1 .. 5) of Integer;
for I in 1 .. 5 loop  Put (Integer'Image (A(I)));
```

Arrays - Range attribute

for I in A'Range loop Put (Integer'Image (A(I)));

- array has an anonymous range for its bounds, since there is no name to refer to the range, Ada solves that via several attributes of array objects
- A'Range = A'First .. A'Last,
- A'First – index of the first element
- A'Last - index of the last element
- A'Length – nr of elements in an array
- "null array", which contains no elements / define an index range whose upper bound is less than the lower bound.

Arrays - Unconstrained arrays

- most powerful aspects of Ada's array
- fixed size: every instance of this type will have the same bounds and therefore the same number of elements and the same size
- bounds are not fixed: the bounds will need to be provided when creating instances of the type

```
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); -- default value
```

Arrays

- Discrete_Type range <> ; -- serves as the type of the index, but different array instances may have different bounds from this type
- unconstrained array type, the bounds need to be provided when an instance is created
- aggregate syntax: associations by name, by giving the value of the index on the left side of an arrow
  association 1=>2 value 2 to element of index 1
- others => 8 --assign value 0 to every element that wasn't previously assigned in this aggregate

Arrays

- "box" notation: (<>) what is expected here can be anything
- unconstrained arrays in Ada, can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value, useless to pass the bounds or length
- different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime

Predefined array type: String

```
type String is array (Positive range <>) of Character;
A : String (1 .. 11) := "Hello World";
B : String (1 .. 11) := ('H', 'e', 'l', 'l', 'o', ' ','W', 'o', 'r', 'l', 'd');
for I in reverse A'Range Put (A (I));
```

- bounds can be omitted when creating an instance of an unconstrained array type if you supply an initialization
- bounds can be deduced from the initialization expression

```
Message : constant String := "dlroW olleH";
```

Arrays

- bounds *have* to be known when instances are created
- can change the values of elements in an array, you cannot change the array's bounds

```
declare
A : String := "Hello";
begin A := "World"; -- OK: Same size
A := "Hello World"; -- Not OK: Different size
end;
```

- violation will generally result in a run-time error
- *indefinite subtype*, which means that the subtype size is not known at compile time, but is dynamically computed

Arrays

- the return type of a function can be any type; a function can return a value whose size is unknown at compile time

function Get_Day_Name (Day : Days := Monday) return String;

- an array of Strings, the String subtype used as component will need to have a fixed size

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

subtype Day_Name is String (1 .. 2); -- Subtype of string with known size

type Days_Name_Type is array (Days) of Day_Name;

Names : constant Days_Name_Type := ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");

Arrays slices

- take and use a slice of an array (a contiguous sequence of elements) as a name or a value
- A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice

```
Buf : String := "Hello ...";
Full_Name : String := "John Smith";
Buf (7 .. 9) := "Bob";
Put_Line (Buf);  --  Prints "Hello Bob"
Put_Line ("Hi " & Full_Name (1 .. 4)); --  Prints "Hi John"
```

Arrays

- objects can be renamed by using the renames
- allows for creating alternative names for these objects
- instead of explicitly using indices every time we're accessing certain positions of the array, we can create shorter names for these positions by renaming them

```
package Measurements is
subtype Degree_Celsius is Float;
Current_Temperature : Degree_Celsius;
end Measurements;
subtype Degrees is Measurements.Degree_Celsius;
T : Degrees renames Measurements.Current_Temperature;
Put_Line (Degrees'Image (T));
```

Records

- Records allow composing a value out of instances of other types
- each of the instances will be given a name
- the pair consisting of a name and an instance of a specific type is called a field, or a component
- record components can have default values
- the value can be any expression of the component type, and may be run-time computable

```
type Date is record Day : Integer range 1 .. 31; Month :
Month_Type := January; -- This component has a default
value
Year : Integer range 1 .. 3000 := 2012; -- Default value end
record;
```

Records - Aggregates

- a convenient notation for expressing values
- called aggregate notation, the literals called aggregates
- used to initalize records
- an aggregate is a list of values separated by commas and enclosed in parentheses
- allowed in any context where a value of the record is expected
- values for the components can be specified positionally or by name
- mixture of positional and named values is permitted, but cannot use a positional notation after a named one

```
Ada_Birthday : Date := (10, December, 1815); Leap_Day_2020 :
Date := (Day => 29, Month => February, Year => 2020);
```

Records - component selection

- to access components of a record instance, we use an operation that is called component selection

```
type Date is record
Day   : Integer range 1 .. 31;
Month : Month_Type;
Year  : Integer range 1 .. 3000 := 2032;
end record;
Some_Day : Date := (1, January, 2000);
Some_Day.Year := 2020;
Put_Line ("Day:" & Integer'Image (Some_Day.Day) & ", Month:
" & Month_Type'Image (Some_Day.Month) & ", Year:" &
Integer'Image (Some_Day.Year));
```

Packages, Generics - intro

Modular programming

- it allows arbitrary declarations in a declarative part
- separation of programs into multiple packages and sub-package
- declare our types and variables in the bodies of main procedures or package specification
- packages let you make your code modular
- with clause indicates a dependency
- accessing entities from a package uses the dot notation
- to make every entity of a package visible directly in the current scope, using use clause

Packages

- **Using a package**
- with Ada.Text_IO; use Ada.Text_IO;
- Put_Line is a subprogram that comes from the Ada.Text_IO
- **Package body**
- Increment_By function has to be declared in the body.
- Last_Increment variable in the body, and make them inaccessible to the user of the Operations package, providing a first form of encapsulation.
- entities declared in the body are *only* visible in the body.

Example

```
package Operations is   --  Declaration
function Increment_By (I: Integer; Incr: Integer := 0) return Integer;
function Get_Increment_Value return Integer;
end Operations;

package body Operations is
Last_Increment : Integer := 1;

function Increment_By (I: Integer; Incr: Integer := 0) return Integer is
begin      if Incr /= 0 then Last_Increment := Incr;      end if;
return I + Last_Increment;   end Increment_By;

function Get_Increment_Value return Integer is
begin      return Last_Increment;   end Get_Increment_Value;
end Operations;
```

Example

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

procedure Main is
I : Integer := 0;
R : Integer;
begin
R := Increment_By (I);
R := Increment_By (I, 10);
end Main;
```

Privacy - Abstract data types

- Encapsulation is the concept that distinguishes between the code's public interface and its private implementation

```
package Stacks is
   type Stack is private; -- You cannot depend on implementation
                          -- You can only assign and test 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;
```

Privacy

- we define a stack type in the public part – visible part
- in the private part, we define the representation of that type
- we can also declare other types that will be used as *helpers* for our main public type
- Stack type as viewed from the public part is called the partial view
- Stack type as viewed from the private part or the body of the package is called the full view of the type
- from the point of view of the client (the *with*'ing unit), only the public (visible) part is important

Usage

```
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

- Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

```
package Stacks is
type Stack is limited private;   --  Cannot assign nor compare.
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

- for some data types the built-in assignment operation might be incorrect
- Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).
- Ada also allows you to implement special semantics for assignment via controlled types
- packages can have child packages

Child packages

- the private part of a package P is meant to encapsulate information,
- certain parts of a child package P.C can have access to this private part of P.
- In those cases, information from the private part of P can then be used as if it were declared in the public part of its specification.
- To be more specific, the body of P.C and the private part of the specification of P.C have access to the private part of P.
- However, the public part of the specification of P.C only has access to the public part of P's specification.

Child packages

```
package Encapsulate is
 procedure Hello;
private
   procedure Hello2;   --  Not visible from external units
                       --  But visible in child packages
end Encapsulate;

with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate is
  procedure Hello is   begin     Put_Line ("Hello");   end Hello;
  procedure Hello2 is   begin     Put_Line ("Hello #2");   end Hello2;
end Encapsulate;

package Encapsulate.Child is
procedure Hello3;
end Encapsulate.Child;
```

Child packages

```
package Encapsulate is
 procedure Hello;
private
   procedure Hello2;   --  Not visible from external units
  --  But visible in child packages
end Encapsulate;

with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate is
  procedure Hello is   begin      Put_Line ("Hello");   end Hello;
  procedure Hello2 is   begin      Put_Line ("Hello #2");   end Hello2;
end Encapsulate;

package Encapsulate.Child is
procedure Hello3;
end Encapsulate.Child;
```

Child packages

```
with Ada.Text_IO; use Ada.Text_IO;
package body Encapsulate.Child is
procedure Hello3 is
begin --  Using private proc Hello2 from the  parent package
 Hello2;   Put_Line ("Hello #3");
end Hello3;
end Encapsulate.Child;

with Encapsulate.Child;
procedure Main is
begin
Encapsulate.Child.Hello3;
end Main;
```

Types in child packages

- same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package.

```
package My_Types is
 type Priv_Rec is private;
private
type Priv_Rec is record
 Number : Integer := 42;
end record;
end My_Types;

package My_Types.Ops is
 procedure Display (E : Priv_Rec);
end My_Types.Ops;
```

Types in child packages

```
with Ada.Text_IO; use Ada.Text_IO;
package body My_Types.Ops is
procedure Display (E : Priv_Rec) is
begin     Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));
end Display; end My_Types.Ops;

with Ada.Text_IO;  use Ada.Text_IO;with My_Types;
use My_Types;with My_Types.Ops; use My_Types.Ops;
procedure Main is
E : Priv_Rec;
begin   Put_Line ("Presenting information:");
-  The following triggers compilation error, no access to Number component
--  Put_Line ("Priv_Rec.Number: " & Integer'Image (E.Number));
Display (E); -- is defined in My_Types.Ops which is child of My_Types,
                -- Ops has access to parent private Priv_Rec
end Main;
```

- allows for extending the functionality of a parent package and,
- at the same time, retain its encapsulation

Generic packages

- Generic keyword and formal specifications

```
generic
   type T is private;
package Element is
procedure Set (E : T);
procedure Reset;
function Is_Valid return Boolean;
private
Value : T;
Valid : Boolean := False;
end Element;
```

Example

```
package body Element is
procedure Set (E : T) is
begin
Value := E;
Valid := True;
end Set;
procedure Reset is
begin
Valid := False;
end Reset;
function Is_Valid return Boolean is (Valid);
end Element;
```

Example

```
with Ada.Text_IO; use Ada.Text_IO; with Element;
procedure Show_Generic_Package is
package I is new Element (T => Integer);
procedure Display_Initialized is
begin
 if      I.Is_Valid then        Put_Line ("Value is initialized");
else        Put_Line ("Value is not initialized");      end if;
end Display_Initialized;
begin
Display_Initialized;   Put_Line ("Initializing...");
I.Set (5);   Display_Initialized;

Put_Line ("Reseting...");   I.Reset;
Display_Initialized;
end Show_Generic_Package;
```

Generic Examples

Generic

- Parametric polymorphism
- code reuse improves the productivity and the quality of software

```
generic
type Element_T is private; -- formal type parameter
procedure Swap (X, Y : in out Element_T);

procedure Swap (X, Y : in out Element_T) is
Temporary : Element_T := X;
begin X := Y; Y := Temporary; end Swap;

procedure Swap_Integers is new Swap (Integer);
procedure Swap_Floats is new Swap (Float);
```

Generic parameters

The generic unit declares *generic formal parameters*, which can be:

- objects (of mode *in* or *in out* but never *out*)
- types
- subprograms
- instances of another, designated, generic unit.
- When instantiating the generic, the programmer passes one *actual parameter* for each formal.
- Formal values and subprograms can have defaults, so passing an actual for them is optional.

Generic formal objects

- Formal parameters of mode *in* accept any value, constant, or variable of the designated type.
- The actual is copied into the generic instance, and behaves as a constant inside the generic;
- the designated type cannot be limited.
- It is possible to specify a default value

```
generic
Object : in Natural := 0;
```

Generic formal types

- The syntax allows the programmer to specify which type categories are acceptable as actuals
- A type declared with the syntax type T (<>) denotes a type with *unknown discriminants*

type T **is private**; -- Any nonlimited definite type, it is possible to assign to variables of this type and to declare objects without initial value

type T **is** (<>); -- Any discrete type: integer, modular, or enumeration.

type T **is range** (<>); -- Any signed integer type

type T **is digits** <>; --Any floating point type

type T (<>) **is private**; Any nonlimited type: the generic knows that it is possible to assign to variables of this type, but it is not possible to declare objects of this type without initial value.

type T (<>) **is limited private**; -- Any type at all. The actual type can be limited or not, indefinite or definite, but the *generic* treats it as limited and indefinite, i.e. does not assume that assignment is available for the type.

Generic formal subprograms

- It is possible to pass a subprogram as a parameter to a generic.
- The actual must match this parameter profile.

```
generic
type Element_T is private;
with function "*" (X, Y: Element_T) return Element_T;
function Square (X : Element_T) return Element_T;

function Square (X: Element_T) return Element_T is
begin
return X * X; -- formal operator "*".
end Square;
```

Generic formal subprograms

```
with Square; with Matrices;
procedure Matrix_Example is
function Square_Matrix is new Square
(Element_T => Matrices.Matrix_T,
 "*" => Matrices.Product);
A: Matrices.Matrix_T:=Matrices.Identity;
begin
A := Square_Matrix (A);
end Matrix_Example;
```

Generic formal subprograms

- It is possible to specify a default with "the box" *is* <>;

```
generic
type Element_T is private;
with function "*" (X, Y: Element_T)
return Element_T is <>;
```

- at the point of instantiation, a function "*" exists for the actual type, and if it is directly visible, then it will be used by default as the actual subprogram.

Generic instances of other generic packages

- A generic formal can be a package; it must be an instance of a generic package, so that the generic knows the interface exported by the package:

```
generic
with package P is new Q (<>);
```

- the actual must be an instance of the generic package Q
- the box after Q means that we do not care which actual generic parameters were used to create the actual for P

Generic package parameter

- It is possible to specify the exact parameters, or to specify that the defaults must be used
- The generic sees both the public part and the generic parameters of the actual package

```
generic
-- P1 must be an instance of Q with the specified actual parameters:
with package P1 is new Q (Param1 => X, Param2 => Y);
-- P2 must be an instance of Q where the actuals are the defaults:
with package P2 is new Q;
```

Instantiating generics

- to instantiate a generic unit, use the keyword **new**:
- the generic formal types define *completely* which types are acceptable as actuals
- Ada requires that all instantiations be explicit.
- it is not possible to create special-case instances of a generic
- the object code can be shared by all instances of a generic
- when reading programs written by other people, there are no hidden instantiations and no special cases

Linear search

- Implement the linear search using generics
- Parameters: element, index, array type and a condition
- The generic should be procedure
- An out parameter should indicate if there is an element of the given condition, and which one is the first

Linear search

```
generic
    type Elem is private;
    type Index is (<>);
    type T is array ( Index range <> ) of Elem;
    with function Prop( A: Elem ) return Boolean;
procedure Linker (x: T; b: out Boolean; j: out Index);
```

Linear search

```
procedure Linker (x: T; b: out Boolean; j: out Index) is
begin
    b:= false;
    for i in reverse x'range loop
        if Prop(x(i)) then b:= true; j:= i; end if;
    end loop;
end linker;
```

Linear search - demo

```
with linker, Ada.Text_IO;
use Ada.Text_IO;
procedure mainlinker is
    type Index is new Integer;
    type Elem is new Integer;
    type T is array (Index range <>) of Elem;
    function myprop (x: Elem) return Boolean is
        begin return (x<0); end myprop;
    k: Index; b: Boolean;
    a: T(1..5):=(1,2,3,4,5);
    a1: T(1..5):=(1,-2,3,-4,5);
    a2: T(1..5):=(1,2,3,4,-5);
procedure Mylinker is new linker (Elem, Index, T, myprop);
```

Linear search demo

```
begin
    mylinker(a, b, k);
    if b then Put_Line( Elem'Image(a(k)) );
        else Put_Line(„ no negativ elements "); end if;
   mylinker(a1, b, k);
   if b then Put_Line( Elem'Image(a1(k)) );
     else Put_Line(" no negativ elements "); end if;
   mylinker(a2, b, k);
   if b then Put_Line( Elem'Image(a2(k)) );
   else Put_Line(" no negativ elements "); end if;
end mainlinker;
```

Conditional maximum search

- Implement the conditional maxumim search
- Parameters: element, index, array type and the searched condition
- The generic should be procedure
- In an out parameter indicate if there is an element of the given condition, and which one is that

Conditional maximum search

```
generic
    type Elem is private;
    type Index is (<>);
    type TA is array (Index range <>) of Elem;
    with function Cond ( A: Elem ) return Boolean;
    with function "<" ( A, B: Elem ) return Boolean is <>;
procedure Max_Search ( T: in TA; V: out Boolean;
                                    Max: out Elem );
```

Max

```
procedure Max_Search ( T: in TA; V: out Boolean;
                                        Max: out Elem ) is
   Mh: Index;
begin
   V := False;
  for I in T'Range loop
     if Cond(T(I)) then
          if V then if T(Mh) < T(I) then Mh := I; end if;
          else V := True; Mh := I; end if; end if; end loop;
   Max := T(Mh);
end Max_Search;
```

Max demo

```
with Max_Search, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use Ada.Integer_Text_IO, Ada.Float_Text_IO;
procedure Max_Demo is
    type T is array (Integer range <>) of Float;
    function Int ( A: Float ) return Boolean is
    begin return A = Float(Integer(A)); end Int;
    procedure Max is new Max_Search(Float, Integer, T, Int);
   A: T(1..10) := (1.4,5.2,3.6,7.0,2.0,65.5,3.0,56.0,2.0,56.0);
   F: Float; V: Boolean;
begin
Max(A,V,F);
if V then Put( F ); end if; end Max_Demo;
```

Map generic

```
generic
    type A is private;
    type B is private;
    type Index is (<>);
    type TA_Array is array ( Index range <> ) of A;
    type TB_Array is array ( Index range <> ) of B;
    with function Op(x: A) return B;
function Map(ta: TA_Array) return TB_Array;
```

Map generic

```
function Map(ta: TA_Array) return TB_Array is
      tb:TB_Array(ta'Range);
begin
for i in ta'Range loop
     tb(i):=op(ta(i));
end loop;
return tb;
end Map;
```

Map demo

```
with map, Ada.Text_IO;
use Ada.Text_IO;
procedure Map_demo is
    type t1 is array (Integer range <>) of Integer;
    type t2 is array (Integer range <>) of Float;
    function square (x: Integer) return Float is
        begin return Float(x*x); end square;
function my_map is new map(Integer, Float, Integer, t1, t2, square);
a: t1(1..5):=(1, 2, 3, 4, 5); b: t2(a'range);

begin b:=my_map(a);
for i in b'Range loop Put_Line(Float'Image(b(i))); end loop;
end Map_demo;
```

Reversal of an array

```
generic
    type Elem is private;
    type Index is (<>);
    type T is array(Index range <>) of Elem;
procedure reversal (a: in out T);
```

Reverse

```
procedure reversal (a: in out T) is
   i: Index:= a'First;
   j: Index:= a'Last;
   tmp : Elem;
begin
    while i<j loop
            tmp:=a(i);
            a(i):=a(j);
            a(j):=tmp;
  i:=Index'Succ(i);
j:=Index'Pred(j);
end loop; end reversal;
```

demo

```
with reversal, Ada.Text_IO; use Ada.Text_IO;
procedure reversalmain is
    type T1 is array (Integer range <>) of Integer;
procedure myreversal is new reversal(Integer, Integer, T1);
a: T1(10..15):=(1,2,3,4,5,6);
a1: T1(10..16):=(1,2,3,4,5,6,7);
a2: T1:=(1,2); a3: T1(1..1); a4: T1(1..0);
begin
myreversal(a);
for i in a'range loop Put_Line(Integer'Image(a(i))); end loop;
end reversalmain;
```

Sort – generic in generic

- Instantiate a generic in another one
- E.g. swap and max_pos should be used in sorting
- Before usage needs instantiation (even if we don't know the types)

Swap generic

```
generic
    type T is private;
procedure Swap ( A, B: in out T );

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

Max_Pos generic function

```
generic
    type Elem is limited private;
    type Index is (<>);
    type TA is array (Index range <>) of Elem;
    with function "<" ( A, B: Elem ) return Boolean is <>;
function Max_Pos ( T: TA ) return Index;
```

Max_Pos generic function

```
function Max_Pos ( T: TA ) return Index is
  Mh: Index := T'First;
begin
   for I in T'Range loop
       if T(Mh) < T(I) then Mh := I;
       end if;
   end loop;
   return Mh;
end Max_ Pos;
```

Generic in generic

```
with Max_Pos, Swap;
procedure Sort ( T: in out TA ) is
  procedure Swap_Elem is new Swap(Elem);
  function Max_Pos_TA is new Max_Pos(Elem, Index, TA);
    Mh: Index;
   begin
       for I in reverse T'Range loop
       Mh := Max_Pos_TA( T(T'First..I) );
       Swap_Elem( T(I), T(Mh) );
       end loop;
end Sort;
```

Sort demo

```
with Ada.Text_IO, Sort; use Ada.Text_IO;
procedure SortDemo is
   type TA is array (Character range <>) of Float;
   procedure R_N is new Sort(Float, Character, TA);
   procedure R_Cs is new Sort(Float, Character, TA, ">");
   T: TA := (3.0,6.2,1.7,5.2,3.9);
begin
   R_Cs(T);
   for I in T'Range loop
      Put_Line( Float'Image( T(I) ) );
   end loop;
end SortDemo;
```

Has repetition

- Implement the `Has_Repetition` generic function with an indefinit vector array type (and its element and index type)
- The function gets a vector and return a boolean value which is true if there is an i such that `v(i) = v(i+1)`
- Test the generic for all possible cases

Has repetition

```
generic
  type Elem is private;
  type Index is (<>);
  type Vector is array ( Index range <> ) of Elem;
function has_repetition( T: Vector) return Boolean;
```

Has repetition

```
function has_repetition( T: Vector) return Boolean is
begin
    if T'length> 1 then
        for i in T'First..Index'Pred(T'Last) loop
            if T(i) = T(Index'Succ(i)) then return True;
            end if;
        end loop;
    end if;
   return False;
end has_repetition;
```

Demo

```
with has_repetition, Ada.Text_IO; use Ada.Text_IO;
procedure demo is
    type TInt is array (Integer range <>) of Integer;
    function my_rep is new has_repetition(Integer, Integer, TInt);
   v1: TInt := (1,1,2,4,5,650);
   v2: TInt := (1,2,3,4,5,6);
   v3: TInt(1..1); --:= (1);
   v4: TInt := (1,2, 3,3,3,56);
   v5: TInt := (1,2, 3,56,56);

begin
v3(1):= 3;
put_line(Boolean'Image(my_rep(v1))); put_line(Boolean'Image(my_rep(v2)));
put_line(Boolean'Image(my_rep(v3))); put_line(Boolean'Image(my_rep(v4)));
put_line(Boolean'Image(my_rep(v5)));
end demo;
```