Dipl. Ing. Mario Blunk

Buchfinkenweg 3
99097 Erfurt / Germany

Phone + 49 (0) 361 6022 5184

Email info@blunk-electronic.de

Web www.blunk-electronic.de

Doc. Vers. 2017-01-24

About Ada

- Programming Language (procedural & object orientated)
- standards *Ada 83, Ada 95, Ada 2005, Ada 2012*
- mission/safety critical applications (aerospace, medical, transportation, financial, military, …)
- **Use it for engineering and commercial !**

- *Ada* is not old !
- *Ada* is not difficult to learn !

For Managers: Why Ada ?

https://en.wikibooks.org/wiki/Ada_Programming :

„… Consequences of these qualities are superior reliability, reusability and maintainability. For example, compared to programs written in C, programs written in Ada 83 contain "70% fewer internal fixes and 90% fewer bugs", and cost half as much to develop in the first place[3] Ada shines even more in software maintenance, which often accounts for about 80% of the total cost of development. ...“

Be courageous !

For Engineers: Why Ada ?

- ***Ada is beautiful !***
- strong and safe **type** system (avoids mixing miles with kilometers, Euros with Rubles, ...)
- **certified compiler** (gnat) detects errors before they become bugs
- reliability – reusability - maintainability
- **A complex world needs a complex language !**

Ready for something new ?

About this Course

- **THIS IS FOR ADVANCED USERS !**
- **We keep things simple !**
- We start with frequently used constructs.
- We learn by many **examples**.
- We focus on **small** command line applications.
- We use very simple *Makefiles.*
- exercises, Q & A

https://github.com/Blunk-electronic/ada_training

Contents

1. environment (arguments, variables, exit status, files & directories)
2. string processing (fixed, bounded, unbounded)
3. parameterized types (discriminants, variants)
4. exceptions (predefined, user specific, handling, propagation)
5. containers (doubly linked lists, vectors, maps)
6. library units (makefile, private types, ...)
7. generics
8. access types (simple, named, not-null, aliased, ...)

https://github.com/Blunk-electronic/ada_training

Command Line Arguments #1

```
with ada.command_line; use ada.command_line;

procedure environment_1 is

begin
   put_line ("argument count: " &
                natural'image(argument_count) );

   put_line ("command: " & command_name);

   if argument_count > 0 then

      for a in 1..argument_count loop
         put_line ( argument(a) );
      end loop;

   end if;
end environment_1;
```

Command Line Arguments #2

```
with ada.command_line; use ada.command_line;

procedure environment_2 is

   i : integer;
   f : float;

begin
   -- i := argument(1); -- does not compile

   i := integer'value( argument(1) );
   put_line("argument 1: " & integer'image(i));

   f := float'value  ( argument(2) );
   put_line("argument 2: " & float'image(f));

end environment_2;
```

Exit Status

```
with ada.command_line; use ada.command_line;

procedure environment_3 is

   e : exit_status := failure;

begin
   if argument_count > 0 then
      put_line ("everything fine");
      e := success;
   else
      put_line ("error: arguments missing !");
   end if;

   set_exit_status (e);

end environment_3;
```

Environment Variables

```
with ada.environment_variables;
use ada.environment_variables;

procedure environment_4 is

begin
   if exists ( "HOME" ) then
      put_line ("my home dir is: " & value("HOME") );
   else
      put_line ("warning: no home directory !");
   end if;

   set ( name => "DUMMY",
         value => "something_meaningful" );

   put_line ("$DUMMY=" & value("DUMMY") );

   clear ("DUMMY");
end environment_4;
```

Directory & File Operations #1

```
with ada.directories; use ada.directories;

procedure directory_and_file_ops_1 is

   dir_name : string (1..3) := "log";

begin
   put_line ("the current working directory is: " &
      current_directory);

   create_directory ( dir_name ); -- created in current
                                  -- working directory
   if exists ( dir_name ) then
      put_line ("directory '" & dir_name & "' created");
   end if;

   delete_directory ( dir_name ); -- clean up

end directory_and_file_ops_1;
```

Directory & File Operations #2

```
with ada.directories; use ada.directories;

procedure directory_and_file_ops_2 is

   file : string (1..19) := "dummy_text_file.txt";
   s : file_size;

begin
   put_line (full_name (file)); -- absolute path
   put_line (containing_directory (file));

   put_line (simple_name (file));
   put_line (base_name (file));
   put_line (extension (file));

   s := size(file);
   put_line (file_size'image(s) & " bytes");

end directory_and_file_ops_2;
```

Directory & File Operations #3

```
with ada.text_io;          use ada.text_io;
with ada.directories;      use ada.directories;

procedure directory_and_file_ops_3 is

   handle : ada.text_io.file_type;
begin
   create (
         file => handle,
         mode => out_file, -- data will go into the file
         name => compose (
               containing_directory => current_directory,
               name => "dummy",
               extension => "txt" )
         );

   put_line ( handle, "This is meaningless stuff.");
   close ( handle );

end directory_and_file_ops_3;
```

Directory & File Operations #4

Read more on functions and procedures in packages:

Ada.Directories
files: a-direct.ads

Ada.Text_IO
files: a-textio.ads

Hint: Find them in system directories like:
/usr/lib/gcc/i586-suse-linux/4.8/adainclude

Fixed Strings #1

```
procedure string_processing_1 is

   a : string (1..3) := "Ada";
   b : string (1..4) := "2005";
   c : string (1..4) := "2012";

begin

   put_line (a & " " & b);
   put_line (a & " " & c);

   b := "2020";

   b := "95"; -- causes a compiler warning
              -- and constraint error at run time

end string_processing_1;
```

Fixed Strings #2

```
with ada.strings.fixed; use ada.strings.fixed;

procedure string_processing_1a is
   a : string (4..10) := "Ada2012";
   b : string (1..20) := (20 * '-');
begin
   put_line (positive'image(a'length));
   put_line (positive'image(a'first));
   put_line (positive'image(a'last));

   put_line (character'image(a(5)));
   a(9) := '2';

   put_line (3 * (a & ' '));

   put_line (b);

end string_processing_1a;
```

Bounded Strings #1

```
with ada.strings.bounded; use ada.strings.bounded;

procedure string_processing_2 is

   package type_universal_string is new
                                 generic_bounded_length(100);
   use type_universal_string;

   a : string (1..3) := "Ada";
   b : type_universal_string.bounded_string :=
                                  to_bounded_string("2005");
begin

   b := to_bounded_string("95");
   put_line (a & " " & to_string(b));

   b := to_bounded_string("2012");
   put_line (a & " " & to_string(b));
end string_processing_2;
```

Bounded Strings #2

```
   a : type_universal_string.bounded_string :=
                        to_bounded_string("Ada2012");

begin
   put_line (positive'image( length(a) ));
   put_line (character'image( element(a,1) ));
   put_line (character'image( element(a, length(a)) ));

   replace_element ( a, 6, '3' );
   put_line (slice (a, 4, 7 ));
   put_line (type_universal_string.to_string(a));

   replace_slice (a, 4, 7, "83" );
   put_line (type_universal_string.to_string(a));
   put_line (natural'image( index(a, "83")) );

end string_processing_2a;
```

Unbounded Strings #1

```
with ada.strings.unbounded; use ada.strings.unbounded;

procedure string_processing_3 is

   a : string (1..3) := "Ada";
   b : unbounded_string := to_unbounded_string("2005");

begin

   b := to_unbounded_string("95");
   put_line (a & " " & to_string(b));

   b := to_unbounded_string("2012");
   put_line (a & " " & to_string(b));

end string_processing_3;
```

Unbounded Strings #2

```
   a : unbounded_string := to_unbounded_string("Ada2012");

begin
   put_line (positive'image( length(a)));
   put_line (character'image( element(a,1) ) );
   put_line (character'image( element(a, length(a)) ) );

   replace_element ( a, 6, '3' );
   put_line (slice (a, 4, 7 ));
   put_line (to_string(a));

   replace_slice (a, 4, 7, "83" );
   put_line (to_string(a));
   put_line (natural'image( index(a, "83")) );

end string_processing_3a;
```

String Type Conversion

```
   package type_us1 is new generic_bounded_length(20);
   use type_us1;
   package type_us2 is new generic_bounded_length(10);
   use type_us2;

   f  : string (1..3) := "Ada";
   b1 : type_us1.bounded_string := to_bounded_string("2005");
   b2 : type_us2.bounded_string := to_bounded_string("2012");
   u  : unbounded_string := to_unbounded_string("95");
begin

   b1 := f & b1;
   -- b1 := b2; -- does not compile
   b1 := to_bounded_string( to_string(b2) );
   put_line (to_string(b1));

   -- u := b1; -- does not compile
   u := to_unbounded_string( to_string(b1) );
   put_line (to_string(u));

end string_processing_4;
```

String Processing Summary

	performance	compatibility	handling
fixed	++	-	--
bounded	+	-	-
unbounded	--	++	++

Parameterized Types #1

```
procedure parameterized_types_1 is

   type type_car ( seat_count : positive ) is
      record
         manufacturer : unbounded_string;
         door_count   : positive;
      end record;

   c : type_car( seat_count => 5 );

begin

   c.manufacturer := to_unbounded_string("Vauxhall");
   c.door_count := 3;

   -- c.seat_count := 4; -- does not compile
   put_line (to_string( c.manufacturer) );

end parameterized_types_1;
```

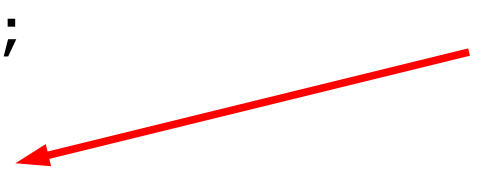
This is a disciminant.

instantiation of a car.

Parameterized Types #2

```
procedure parameterized_types_2 is
   type type_car ( seat_count : positive) is
      record
         manufacturer : unbounded_string;
         door_count   : positive;
         case seat_count is
            when 1..8 => null;
            when others =>
               special_driving_license : unbounded_string;
         end case;
      end record;

   c : type_car( seat_count => 4 );
begin
   c.manufacturer := to_unbounded_string("Vauxhall");
   c.special_driving_license :=
                  to_unbounded_string("class 1");
                  -- causes a warning at compile time
                  -- and constraint error at run time
end parameterized_types_2;
```



Exceptions

When something goes wrong …

- *invalid/missing arguments*
- *array index out of range*
- *failed file access*
- *insufficient memory*
- *division by zero*
- *...*

Exceptions: Constraint_Error #1

```
with ada.text_io; use ada.text_io;

procedure exceptions_1 is

   type array_of_integers is array (positive range 1..5)
                                                 of integer;
   a : array_of_integers;

begin

   a(6) := 4; -- Assign non-existing member 6 the value 4.

   -- Causes a warning at compile time and
   -- a CONSTRAINT_ERROR at run time.

end exceptions_1;
```

Exceptions: Constraint_Error #2

```
procedure exceptions_2 is

   type array_of_integers is array (positive range 1..5)
                                                  of integer;
   a : array_of_integers;
begin
   a(6) := 4; -- Assign non-existing member 6 the value 4.
   -- Causes a warning at compile time and
   -- a CONSTRAINT_ERROR at run time.
   -- Program control passed to exception handler.

   put_line("Everything fine."); -- skipped on exception

   -- Exception handler:
   exception
      when constraint_error =>
         put_line ("ERROR: Array index invalid !");

end exceptions_2;
```

Exceptions: Constraint_Error #3

```
procedure exceptions_3 is

   p, q, r : natural := 0;

begin
   r := p / q; -- Division by zero.
   -- Causes a warning at compile time and
   -- a CONSTRAINT_ERROR at run time.
   -- Program control passed to exception handler.

   put_line("Everything fine."); -- skipped on exception

   -- Exception handler:
   exception
      when constraint_error =>
         put_line ("ERROR: Division by zero !");

end exceptions_3;
```

Predefined Exceptions

predefined exceptions:

- *Constraint_Error (frm. Numeric_Error)*
- *Program_Error*
- *Storage_Error*
- *Tasking_Error*

Exception Handler #1

```
procedure exceptions_4 is
   data_format_error : exception; -- user specific exception
   operator_error    : exception; -- user specific exception
begin
   -- We intentionally raise exceptions to demonstrate the
   -- exception handler:

   -- raise constraint_error;
   -- raise storage_error;
   -- raise data_format_error;
   raise operator_error;

   exception
      when constraint_error =>
         put_line ("Constraint error occured !");
      when storage_error =>
         put_line ("Storage error occured !");
      when data_format_error =>
         put_line ("Data format error occured !");
      when others =>
         put_line ("Other error occured !");
end exceptions_4;
```

Exception Handler #2

```
procedure exceptions_5 is
   operator_error    : exception; -- user specific exception
   program_position  : natural := 0;
begin
   --raise operator_error;
   program_position := 10;  --raise operator_error;
   program_position := 30;  raise operator_error;
   put_line("Everything fine."); -- skipped on exception

   exception
      when constraint_error =>
         put_line ("Constraint error occured !");
      when operator_error =>
         put ("Operator error ! ");
         case program_position is
            when  0 => put_line ("Missing arguments");
            when 10 => put_line ("Invalid argument given.");
            when others => put_line ("Contact administrator !");
         end case;
      when others =>
         put_line ("Other error occured !");
end exceptions_5;
```

Exception Handler #3

```
with ada.exceptions; use ada.exceptions;

procedure exceptions_6 is
   operator_error    : exception; -- user specific exception
begin

   raise operator_error with "Wrong key pressed !";
   --raise constraint_error;

   put_line("Everything fine."); -- skipped on exception

   exception
      when event:
         operator_error =>
            put_line(exception_information(event));
      when constraint_error =>
         put_line ("Constraint error occured !");
      when others =>
         put_line ("Other error occured !");

end exceptions_6;
```

Exception Handler #4

see more advanced example at github

https://github.com/Blunk-electronic/ada_training:

exceptions_6a.adb

Exception Propagation

```
procedure exception_prop_1 is
   operator_error    : exception; -- user specific exception

   procedure request_operator_input is
   begin
      null; -- assume operator input here
      raise operator_error; -- we intentionally raise an
                            -- exception
   end request_operator_input;

begin
   request_operator_input;

   put_line("Everything fine."); -- skipped on exception

   exception
      when operator_error =>
         put_line ("Operator error occured !");
      when others =>
         put_line ("Other error occured !");

end exception_prop_1;
```

Containers

When objects are to be stored, sorted and queried ...

- *doubly linked lists* (objects accessed by just a cursor, no indexing)
- vectors (objects accessed by both a cursor or an index)
- ordered maps (objects accessed by both a cursor or a key)

The advantage of containers over arrays:

Objects can be added and removed freely without having to care about any bounds.

Doubly Linked Lists #1

```
with ada.containers; use ada.containers;
with ada.containers.doubly_linked_lists;

procedure cont_doubly_linked_list_1 is
   package type_my_list is new doubly_linked_lists(natural);
   l : type_my_list.list;
   c : type_my_list.cursor;
   n : natural;
begin

   type_my_list.append(l,7); -- append object '7' to list 'l'
   type_my_list.append(l,9); -- append object '9' to list 'l'

   c := type_my_list.first(l);   -- set cursor at begin of list
   n := type_my_list.element(c); -- get first object
   put_line(natural'image(n));   -- display object

   c := type_my_list.next(c);    -- advance cursor to next object
   n := type_my_list.element(c); -- get next object
   put_line(natural'image(n));   -- display object

end cont_doubly_linked_list_1;
```

Doubly Linked Lists #2

Read more in package specification:

Ada.Containers.Doubly_Linked_Lists
file: a-cdlili.ads

Hint: Find it in system directories like:
/usr/lib/gcc/i586-suse-linux/4.8/adainclude

Vectors #1

```
with ada.containers; use ada.containers;
with ada.containers.vectors;

procedure cont_vectors_1 is

   package type_my_vector is new
      vectors (  index_type => positive,
                 element_type => natural);

   v : type_my_vector.vector;
   n : natural;

begin

   type_my_vector.append(v,7); -- add first object
   type_my_vector.append(v,9); -- add next object

   n := type_my_vector.element(v,2); -- get object from pos. 2
   put_line(natural'image(n));

end cont_vectors_1;
```

Vectors #2

Read more in package specification:

Ada.Containers.Vectors
file: a-convec.ads

Hint: Find them in system directories like:
/usr/lib/gcc/i586-suse-linux/4.8/adainclude

Ordered Maps #1

```
with ada.containers; use ada.containers;
with ada.containers.ordered_maps;

procedure cont_maps_1 is

   package type_my_map is new
      ordered_maps (  key_type => positive,
                    element_type => natural);

   m : type_my_map.map;
   n : natural;
begin
   type_my_map.insert(m,123,7); -- ins. object '7' with key '123'
   type_my_map.insert(m,788,99); -- ins. obj. '99' with key '788'

   n := type_my_map.element(m,788); -- get object with key  '788'
   put_line(natural'image(n));

   n := type_my_map.element(m,123); -- get object with key '123'
   put_line(natural'image(n));

end cont_maps_1;
```

Ordered Maps #2

```
with ada.containers; use ada.containers;
with ada.containers.ordered_maps;

procedure cont_maps_2 is

   package type_my_map is new
      ordered_maps (  key_type => character,
                    element_type => natural);

   m : type_my_map.map;
   n : natural;
begin
   type_my_map.insert(m,'A',7); -- ins. object '7' with key 'A'
   type_my_map.insert(m,'X',99); -- ins. obj. '99' with key 'X'

   n := type_my_map.element(m,'X'); -- get object with key  'X'
   put_line(natural'image(n));

   n := type_my_map.element(m,'A'); -- get object with key 'A'
   put_line(natural'image(n));

end cont_maps_1;
```

Ordered Maps #2

Read more in package specification:

Ada.Containers.Ordered_Maps
file: a-coorma.ads

Hint: Find them in system directories like:
/usr/lib/gcc/i586-suse-linux/4.8/adainclude

Library Units: Why ?

When code is to be re-used !

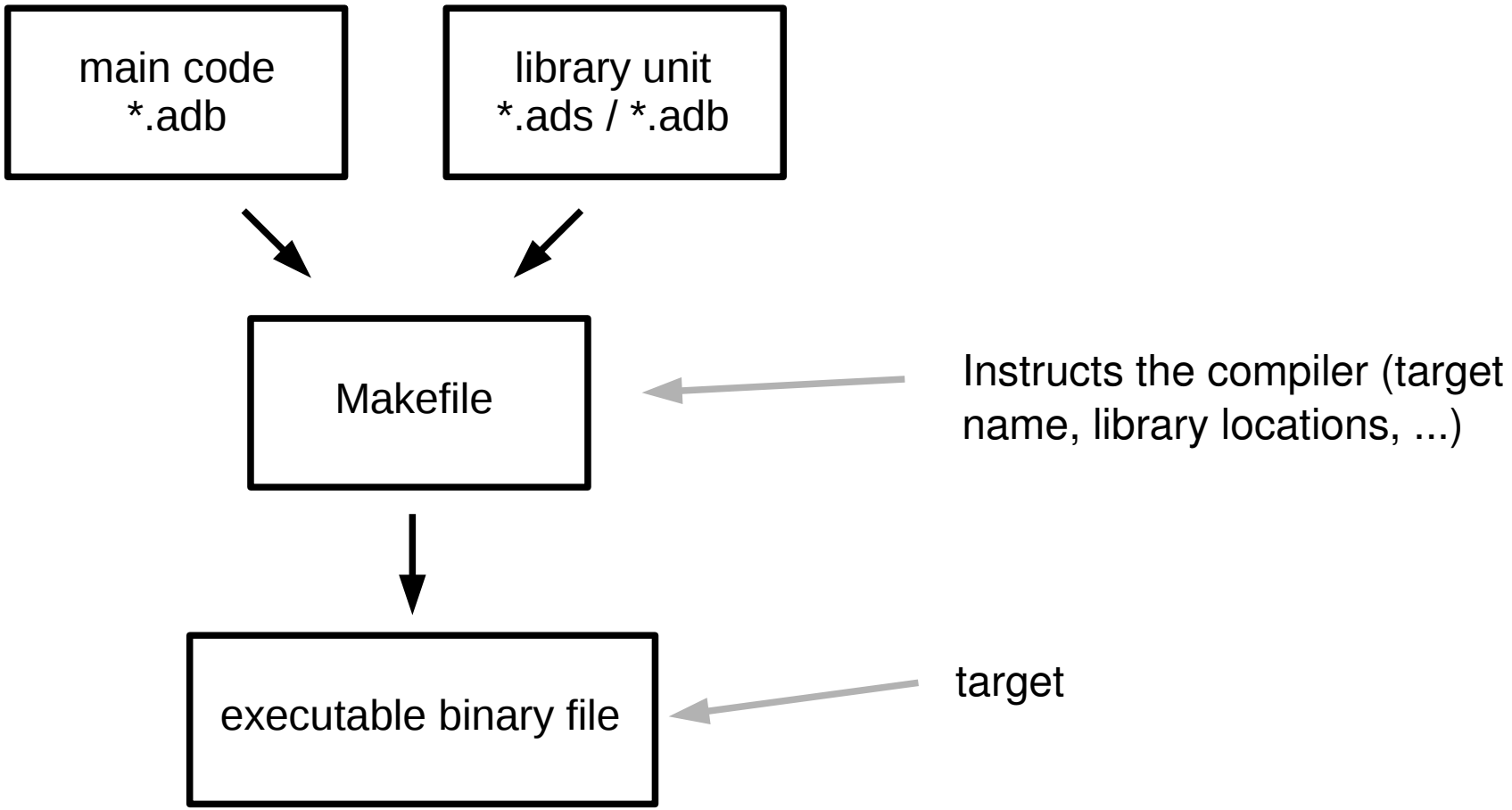
- Avoid writing the same code more than once !
- Use library units (also called packages) !

An Ada library unit consists of two files:

*specification: *.ads* *body: *. adb*

Library Units: Make

The compile process:



Library Units: Specification

```
------------------------------
--                          --
--      simple_library      --
--                          --
--         S p e c          --
------------------------------

package simple_library is

   procedure output_text (text : in string);

end simple_library;
```

Library Units: Body

```
------------------------------
--                          --
--      simple_library      --
--                          --
--         B o d y          --
------------------------------

with ada.text_io; use ada.text_io;

package body simple_library is

   procedure output_text (text : in string) is
   begin
      put_line ( text );
   end output_text;

end simple_library;
```

Library Units: Parent

```
with simple_library; -- use simple_library;

procedure library_units_1 is

begin

   simple_library.output_text ("hello world !");

end library_units_1;
```

Library Units: Include

```
$> ls -l
total 8
lrwxrwxrwx 1 hans users   6 2017-01-06 09:08 include -> ../lib
-rw-r--r-- 1 hans users 244 2017-01-06 09:08 library_units_1.adb
-rw-r--r-- 1 hans users 392 2017-01-06 09:14 Makefile
```

Hint: create symbolic UNIX/Linux link:

ln -s target target_name

example command here: **`ln -s ../lib include`**

Private Types

When internals of objects are to be hidden.

- need-to-know principle
- coding safety

Private Types: Why ?

```
procedure types_private_1 is

   type wealth is record
      cash, estate, total : natural := 0;
   end record;

   function "+" (a,b : in wealth) return wealth is
      c : wealth;
   begin
      c.cash := a.cash + b.cash;
      c.estate := a.estate + b.estate;
      c.total := c.cash + c.estate;
      return (c);
   end "+";

   w, p : wealth;
begin
   p.cash := 8;  p.estate := 4;  w := w + p;
   put_line ("total  :" & natural'image(w.total));
   w.total := 100; -- Compiles, yet it's a lie !

end types_private_1;
```

Private Types: Specificaton

```
package library_with_private_types is

   type wealth is private;

   function take_money  (c : in positive) return wealth;
   function take_estate (e : in positive) return wealth;

   function "+" (a,b : in wealth) return wealth;

   function show_wealth (w : in wealth) return natural;

   private
      type wealth is record
         cash   : natural := 0;
         estate : natural := 0;
         total  : natural := 0;
      end record;

end library_with_private_types;
```

Private Types Body

Find package body of „library_with_private_types“ here.

Due to its size it does not fit here.

Private Types: Parent

```
with library_with_private_types; use library_with_private_types;

procedure types_private_2 is

   w, p : wealth;

begin
   w := take_money(8);
   p := take_estate(4);
   w := w + p;

   put_line ("total  :" & natural'image(show_wealth(w)));

   -- w.total := 100; -- does not compile

end types_private_2;
```

Generics

When code is to be re-used !

- equal handling of object regardless of their type
- generic subprograms
- Use generic packages !

An Ada generic package usually consists of two files:

*specification: *.ads* *body: *. adb*

Generics #1

```
procedure generics_1 is

   generic -- subprogram specification
      type item is private;
   procedure swap_items(x,y : in out item);

   procedure swap_items(x,y : in out item) is -- subprogrm body
      scratch : item := x;
   begin
      x := y;    y := scratch;
   end swap_items;

   procedure swap_natural is new swap_items(item => natural);
   x : natural := 10;
   y : natural :=  7;

begin
   put_line("x:" & natural'image(x) & "  y:" & natural'image(y));
   swap_natural(x,y);
   put_line("x:" & natural'image(x) & "  y:" & natural'image(y));

end generics_1;
```

Generics as Package: Spec

```
------------------------------
--                          --
--      library_with_generic   --
--                          --
--         S p e c          --
------------------------------

package library_with_generic is

   generic
      type item is private;
   procedure swap_items(x,y : in out item);

end library_with_generic;
```

Generics as Package: Body

```
----------------------------------
--                              --
--      library_with_generic    --
--                              --
--          B o d y             --
----------------------------------

package body library_with_generic is

   procedure swap_items(x,y : in out item) is
      scratch : item := x;
   begin
      x := y;    y := scratch;
   end swap_items;

end library_with_generic;
```

Generics as Package: Parent #1

```
with library_with_generic;

procedure generics_2 is

   use library_with_generic;
   procedure swap_natural is new swap_items(item => natural);

   x : natural := 10;
   y : natural :=  7;

begin

   put_line("x:" & natural'image(x) & "  y:" & natural'image(y));

   swap_natural(x,y);

   put_line("x:" & natural'image(x) & "  y:" & natural'image(y));

end generics_2;
```

Generics as Package: Parent #2

```
with ada.strings.unbounded; use ada.strings.unbounded;
with library_with_generic;

procedure generics_3 is

   use library_with_generic;
   procedure swap_unbounded is new swap_items
                  (item => unbounded_string);

   x : unbounded_string := to_unbounded_string("ABC");
   y : unbounded_string := to_unbounded_string("XYZ");

begin

   put_line("x:" & to_string(x) & "  y:" & to_string(y));

   swap_unbounded(x,y);

   put_line("x:" & to_string(x) & "  y:" & to_string(y));

end generics_3;
```

Access Types

When objects exist independed of the program structure.

- creating, modifying objects
- in a storage pool (heap)
- objects residing there are accessed by Access Types
- An Access Type can access **only** a defined type of object !
- An Access Type is more than just a memory address !
- more flexibility !

Access Types: Basics #1

```
with ada.text_io;  use ada.text_io;

procedure types_access_1 is

   -- Declare an access to an integer:
   ai : access integer;

begin

   -- Create an integer of value -10 where ai is pointing at:

   ai := new integer'(-10);


   -- ai := new float'(1.5); -- does not compile

   -- Display the value of the integer where ai is pointing at:
   put_line ( integer'image(ai.all) );

end types_access_1;
```

allocator with init value

Access Types: Basics #2

```
procedure types_access_2 is

   -- Declare two accesses to an integer:
   ai, bi : access integer;

begin
   -- Create integer #1 of value -10 where ai is pointing at:
   ai := new integer'(-10);
   bi := ai; -- backup address of integer #1

   -- Create integer #2 of value 4 where ai is pointing at now:
   ai := new integer'(4);

   -- Display the value of the integer where ai is pointing at:
   put_line ( integer'image(ai.all) );

   ai := bi; -- restore address of integer #1

   -- Display the value of the integer where ai is pointing at:
   put_line ( integer'image(ai.all) );

end types_access_2;
```

Access Types: Basics #3

```
procedure types_access_3 is

   -- Create two different incompatible named access types:
   type ptr_a is access integer;
   type ptr_b is access integer;

   -- Declare two different accesses to an integer:
   ai : ptr_a;
   bi : ptr_b; -- try ptr_a ?

begin
   -- Create integer #1 of value -10 where ai is pointing at:
   ai := new integer'(-10);
   -- Create integer #2 of value -20 where ai is pointing at:
   ai := new integer'(-20);
   -- Change integer #2 to value -21
   ai.all := -21;

   -- bi := ai; -- does not compile

   put_line ( integer'image(ai.all) );
end types_access_3;
```

Access Types: Basics #4

```
procedure types_access_4 is

   -- Create two different incompatible named access types:
   type ptr_a is access integer;
   type ptr_b is access integer;

   -- Declare two different accesses to an integer:
   ai : ptr_a;
   bi : ptr_b;

begin
   -- Create integer #1 of value -10 where ai is pointing at:
   ai := new integer'(-10);
   -- Create integer #2 of value -20 where ai is pointing at:
   bi := new integer'(-20);

   -- Overwrite integer #1 with integer #2 (copy):
   ai.all := bi.all;

   put_line ( integer'image(ai.all) );

end types_access_4;
```

Access Types: Basics #5

```
procedure types_access_5 is

   type ptr_a is access integer;
   type ptr_b is access integer;

   ai : ptr_a;   bi : ptr_b; -- Both point nowhere ! Like
                             -- default:
   -- ai : ptr_a := null;
   -- bi : ptr_b := null;

begin

   -- Create integer #1 of value -10 where ai is pointing at:
   ai := new integer'(-10);

   ai.all := bi.all; -- Causes warning at compile and
                     -- constraint error at run time.

   put_line ( integer'image(ai.all) );

end types_access_5;
```

Because we can't copy nothing to somewhere !

Access Types: Basics #6

```
procedure types_access_6 is

   type ptr_a is not null access integer;
   type ptr_b is not null access integer;

   -- ai : ptr_a; -- Causes a warning at compile and a
                  -- constraint error at run time.

   -- Allocate and initialize integer #1 accessed by ai:
   ai : ptr_a := new integer'(-10);
   -- Allocate and initialize integer #2 accessed by bi:
   bi : ptr_b := new integer'(-20);

begin

   -- Overwrite integer #1 with integer #2
   ai.all := bi.all;

   -- Display the value of the integer where ai is pointing at:
   put_line ( integer'image(ai.all) );

end types_access_6;
```

Enforced initialization !

Access Types: Basics #7

```
procedure types_access_7 is

   -- Declare and initialize an aliased integer i:
   i : aliased integer := 10;

   -- Define a general access to an integer type:
   type ptr is access all integer;
   p : ptr;

begin
   put_line ( integer'image (i) ); -- i before manipulation

   p := i'access; -- p assumes address of i

   p.all := 20; -- assign new value where p points at

   put_line ( integer'image (i) ); -- i after manipulation

   -- put_line ( integer'image (p.all) ); -- display by reference

end types_access_7;
```

Access to Records #1

```
procedure access_records_1 is

   -- Define an apple:
   type apple is record
      weight : float;
      size   : float;
      rotten : boolean;
   end record;

   -- Define an access to record type "apple"
   type ptr is access apple;

   -- Allocate and initialize an apple accessed by pa:
   pa : ptr := new apple'(weight => 0.23, size => 6.4,
                                          rotten => false);
begin

   -- Display the weight of the apple accessed by pa:
   put_line ( float'image(pa.weight) );

end access_records_1;
```

Access to Records #2

```
procedure access_records_2 is

   type apple is record
      weight : float;
      size   : float;
      rotten : boolean;
   end record;

   -- Define two incompatibe access types to record type "apple"
   type ptr_a is access apple;
   type ptr_b is access apple;

   -- Allocate and initialize two apples accessed by pa and pb:
   pa : ptr_a := new apple'(weight => 0.23, size => 6.4,
                                            rotten => false);
   pb : ptr_b := new apple'(weight => 0.4, size => 5.9,
                                            rotten => true);
begin
   pa.all := pb.all; -- copy objects from pb to pa

   put_line ( float'image(pa.weight) );
end access_records_2;
```

Access to Procedures #1

```
procedure access_procedures_1 is

   -- Define an access to ANY procedure that takes a string:
   type type_my_access is not null access
                                procedure (s : in string);

   -- This is a simple procedure which takes a string:
   procedure say_hello ( text : in string ) is begin
      put_line(text);
   end say_hello;

   -- Instantiate an access of type type_my_access that refers to
   -- procedure say_hello.
   p : type_my_access := say_hello'access;

begin
   -- Call procedure say_hello via access p:
   p("hello");

end access_procedures_1;
```

Access to Procedures #2

```
procedure access_procedures_2 is
   -- Define an access to ANY procedure that "inouts" a number:
   type type_my_access is not null access
                        procedure (n : in out integer);

   procedure double (x : in out integer) is begin
      x := x * 2;
   end double;

   procedure square (y : in out integer) is begin
      y := y ** 2;
   end square;

   a : integer := 3;
   p : type_my_access := double'access;

begin
   p(a); put_line(integer'image(a)); -- result 6
   p := square'access;
   p(a); put_line(integer'image(a)); -- result 36

end access_procedures_2;
```

Access to Functions #1

```
procedure access_functions_1 is

   -- Define an access to ANY function takes and returns an
   -- integer:
   type type_my_access is access
                  function ( i : in integer) return integer;

   -- This is a simple function that takes and returns an
   -- integer:
   function double ( n : in integer ) return integer is begin
      return n * 2;
   end double;

   -- Instantiate an access of type type_my_access that refers to
   -- function double.
   p : type_my_access := double'access;

begin
   -- Call function double via access p:
   put_line( integer'image( p(4) ) );

end access_functions_1;
```

Access to Functions #2

```
procedure access_functions_2 is

   -- Define an access to ANY function takes and returns an integer:
   type type_my_access is access
                       function ( i : in integer) return integer;

   function double ( n : in integer ) return integer is begin
      return n * 2;
   end double;

   function square ( n : in integer ) return integer is begin
      return n ** 2;
   end square;

   p : type_my_access := double'access;

begin
   put_line( integer'image( p(4) ) ); -- result 8
   p := square'access;
   put_line( integer'image( p(4) ) ); -- result 16

end access_functions_2;
```

References

(1)http://www.adaic.org

(2)https://en.wikibooks.org/wiki/Ada_Programming

(3)https://github.com/Blunk-electronic/ada_training

(4)http://www.blunk-electronic.de/ada.html

(5)Barnes, John; Programming in Ada 20xx;

Thanks for your attention !