Introduction to *Ada95*

- Types and program flow control
- Compilation units, ”packages”
- Tasks in Ada95
- Protected objects in Ada95
- Example from Exercise Compendium

A simple example...

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

procedure hello is
begin
   put_line( ”Hello”);
end;
```

Compare:

Java: `'import'`,

C: `"include"`

Source code file has extension `.adb`

-> `hello.adb`

Compile *and* link with: `gnatmake hello.adb`

or

Compile separately with : `gada68 hello.adb`

and then linked with: `gbind68 hello`

And run : `./hello`

Identifiers

Identifiers

- Can be of arbitrary length
- Can only contain characters, digits and underscore (’_’)
- Must start with a character

```
EXAMPLE:
hello
Minimum_Delay
Minimum_Delay_2
A_Very_Verbal_Identifier_Name_Without_Useful_Information
```


Literals

Literals

are *character representation* of numerical values. May also include information about the numeric radix (base of the value).

EXAMPLE:

`10000` same as `10_000`	type: ”universal_integer”
`3.1416`	type: ”universal_float”
`3E6` I.e. 3×10^6	type: ”universal_float”
`16#FE12#`	constant, radix 16
`'a', 'c'`	character constant
`"hello"`	string constant

Reserved words

```
abort   case    for     new     raise   tagged
abs     constant        function        not
range   task    abstract        null    record
```

etc, etc...

Reserved words cannot be used as identifiers...

Types

Scalar

Character: 8 bits (Latin-1)

Wide_Character: 16 bits.

Boolean: can be TRUE or FALSE

(se ADA Distilled, A1)

Float: Real numbers.

Integer: Integer numbers

Enumeration type, EXAMPLE

```
type enumerate (first,second,third);
```

Composite

Array, arbitrary dimension.

EXAMPLE:

```
type mystring is array(1..16)
   of character;
```

EXAMPLE

Record:

```
type imnum is record
   Real_Part   : float;
   Im_Part     : float;
end record;
```


More on integer types...

EXAMPLE declarations:

```
x,y,z  : integer;                  -- vaiables, undefined contents
length : integer := 17;            -- declaration and initialisation
month  : integer range 1..12;      -- with range constraints
```

alternative:

```
subtype months is integer range 1..12;   -- a new type
month  : months;
```


Integer operators

+	addition
–	subtraction
/	integer division, `A/B = c + d/e`
*	multiplication
**	exponent, EXAMPLE `y**x` (”y raised to x”)
mod	modulus result from division, `A mod B = c + d/e`
rem	modulus result from division, `A mod B = c + d/e`
	note: `mod/rem` result may differ depending on A and B signs
abs	absolute value

Simple ”array” type

```
type name is array(startindex..stopindex) of element_type;
```

EXAMPLE:

```
type stack is array(1..50)of integer;
ws    :stack;     -- variable 'ws' of type 'stack'
ws:=(1=>10,2=>20,3=>30, others => 0 );
```

can also be written...

```
ws:=(10,20,30, others => 0 );
```


Unlimited ”array” type

```
type name is array (index_type range <>) of element_type;
```

EXAMPLE:

```
type num_vec is array(integer range <>)of float;
vector1      : num_vec(-10..10);
n:           : integer := 8;
vector2      : num_vec(1..n);

for i in vektor2'range loop
      null;
end loop;
```

attribute

MULTI DIMENSION ARRAY EXAMPLE:

```
type matrix5 is array(1..4,1..10,-8..16,32..36,1..8) of
integer;
```


Composite type ”record”

```
type name is record
       -- simple types, arrays
end record;
```

Records can be expanded with records, EXAMPLE:

```
type info is tagged record
       name : string(1..64);
       age  : integer;
end record;
```

```
type add_info is new info with
record
       male : boolean;
end record;
```


Pointers

The name of a pointer in Ada95 is **access type**. Type checking is strong. E.g. `access type INTEGER` is not the same as `access type CHARACTER`

```
type PtrTo_info is access all info;
ptr     : PtrTo_info;
ptr_2   : PtrTo_info;
       ptr := new info( "Lena ", 35 );
       ptr_2.all := ptr;       -- entire record copied
       ptr_2.age = 36;         -- record member initialized
```

Distinguish between the pointer value, which is an address, and the object that the pointer points to. The pointer value is accessible with the attribute `access`.

```
ptr_2 :=       ptr'access;
```

`ptr_2` and `ptr` now addresses the same object.

More on pointers...

Attribute `all` can only be used with acces types.

Uninitialised acces type objects has the value `null`.

A pointer declared `access all` can hold the address of a *static object*.

In this case, the object has to be declared `aliased`.

```
object  : aliased info
ptr     : PtrTo_info;
begin
        ptr = object'access;
        ...
```

Section 5 in "ADA Distilled"

Program flow control

if/else

```
if boolean expression then
      statement(s);
else
      statement(s);
end if;
```


Program flow control

if/elsif

```
if boolean expression then
      statement(s);
elseif another boolean expression then
      statement(s);
else
      statement(s);
end if;
```


Program flow control

case

```
case  letter is
      when 'a'..'z' => put( "small");
      when 'A'..'Z' => put( "capital");
      when '0'..'9' => put( "number");
      when others => put( " don't know...");
end case;
```

Program flow control

”do while”

```
loop
      statement(s);
      exit when boolean expression ;
end loop;
```


Program flow control

”while do”

```
while boolean expression loop
      statement(s);
end loop;
```


Program flow control

”for...”

```
for i in (reverse) startvalue..stopvalue
loop
      statement(s);
end loop;
```

Loop variable `i` is not excplicitly declared so it cannot be modified by the program. It is assigned *`startvalue`* prior to the first iteration, then increased by 1 for each iteration.

If the form ”`reverse`” is used, the loop variable is initialised with *`stopvalue`*, then decremented by 1 for each iteration.

Program block

Blocks is used to define the first and the last execution point in a program. Blocks may nests.

A block is always started with:

```
begin
-- statements and program flow control
```

and ended with:

```
end
```

Subprograms

Only *functions* and *procedures* can be defined as subprograms. They must always have an executable block.

A function always returns a value, it can (but should not) have side effects. Values can be passed to functions.

A procedure is the general execution unit, values can be passed to procedures as ”in”, ”out” or ”in/out”.

Subprogram ”function”

EXAMPLE: Function `even` returns `TRUE` if parameter is even, otherwise returns `FALSE`.

```
function even( num: in integer) return boolean is
begin
        if num mod 2 = 0 then
                return TRUE;
        else
                return FALSE;
        end if;
end;
```

Parameter can be `in` or `access`. A value should always be returned and this is checked at compile time.

Subprogram ”procedure”

```
procedure even( num: in integer, res: out boolean) is
begin
        if num mod 2 = 0 then
                res = TRUE;
        else
                res = FALSE;
        end if;
end;
```

Parameter can be `in`, `out`, `inout` or `access`. There are no return values.

Overloading: Two subprograms can have the same identifiers (names) presumed that their parameter lists are different.

Generic units

- Generic routines are general descriptions of algoritms that can be used with different data types.
- An instansiation, where a desired type is specified, creates the usable object(s)..

Declaration

```
package generic_stack is
        generic
        type element is private;
  package stack is
        procedure push(e: element);
        procedure pop (e: element);
  end stack;
end;
```

Instansiation

```
with    generic_stack;
use     generic_stack;
procedure stacktest is
  package int_stack
        is new stack(integer);
begin
        ...
        stack.push( data );
end;
```

Unchecked type conversions

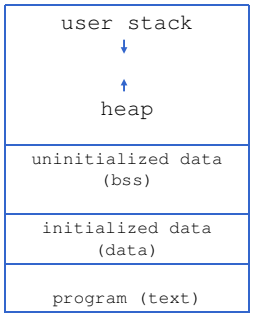
Generic standard procedure `unchecked_conversion`
EXAMPLE: Convert access type to integer

```
with    unchecked_conversion;
procedure unckecked is
  type segment is array(1..1023) of character;
  type pointer is access all segment;
  memory       : aliased segment;
  new_memory   : pointer;
  start_address: integer;

  -- a new instance ...
  function to_int is new unchecked_conversion(pointer,integer);

begin
       new_memory := memory'access;
       start_address := to_int( new_memory );
end;
```


The Run-Time Environment



`stack`: local variables, temporary save of return address during subprogram calls.

`heap`: used for run-time (dynamic) memory allocation (operator **`new`**).

stack grows downwards while *heap* grows upwards in memory.

`bss`: ("block started by symbol") non- initialised memory, used for global variables.

`data`: global initialised variables.

`text`: devoted to machine instructions. Only executable segment.

Somewhat simplified...

Tasks in Ada95

A "subprogram" executing in parallel with other subprograms is called a *task*.

Liza and John goes shopping *simultanously*.

```
procedure shopping  is
       task   john;
       task   liza;
task body john is
begin
       Get_Bread;
end john;

task body liza is
begin
       Get_Milk;
end liza;

begin
       null;
end    shopping;
```


Task synchronisation

- Tasks are synchronised by mean of *rendezvous*.
- A rendezvous specification is called an *entry*.
- Entries are the only allowed declarations in a task specification.

```
task p is
       entry e1( i : in integer );
       entry e2;
end;
```

Entries can be called (requesting a rendezvous) from another subprogram:

```
       ...
       p.e1( n );
       ...
       p.e2;
       ...
```

Task synchronisation, continued...

For each entry there must be at least one *accept*-statement

```
task body p
...
       accept e1( i : in integer )
       do
              ...
       end e1;
...
```

When **p** reaches `accept` the queue for this entry (`e1`) is checked. If the queue is non-empty, i.e. any task **q** has called the entry (`p.e1()`) a *rendezvous* is undertaken. The block within do-end is then executed, without interruption (critical region).
If **p** reaches accept- prior to a call to this entry, **p** is blocked and thus cannot proceed ececution.

Task synchronisation, continued...

- A single entry can be called from any subprogram.
- Each entry call is queued (FIFO).
- Each call is granted a rendezvous that cannot be interrupted.
- Each time the accept statement is reached, the corresponding queue is examined for pending entry calls.
- A task cannot be a library unit, it must reside as a subprogram.

Protected objects

- A protected object insures mutual exclusion. There is always at least one variable, that has to be *protected*.

- A protected object package includes procedures and functions that can be called anytime, but may block depending on run-time circuimstances. Typically, these procedures/functions manipulates the protected variable(s).

- A protected object is used to manage an exclusive resource. There are similarities between tasks and protected objects, but they are not quite the same. An entry can ”call itself” (requeue) but a protected object member cannot call itself, another member of the object OR another protected object.

Protected objects, EXAMPLE

```
with Ada.Text_IO;
        procedure Protected_Variable_Example is
        protected Variable is
                procedure Modify(Data : Character);
                -- Object 'Variable' is locked for this operation

                function Query return Character ;
                -- Read-only. May not update data

                entry Display(Data : Character; T : String);
                -- An entry has a queue

                private
                        Shared_Data : Character := '0';
                        -- All data is declared here
        end Variable;

        protected body Variable is -- No begin end part in protected body
```

ADA Distilled 14.3

Exceptions

Exceptional events are things that happens, during program execution, that prevents further execution of a subprogram.
EXAMPLES:

- Divide by 0
- Assigning values out of bound
- Requesting memory beyond limits
- etc. etc.

Ada provides mechanisms for the application programmer to handle such events.

This will be further elaborated throughout this course

Tools used in this course:

GPS, for Windows (or similar for Linux)

General Ada programming, work at home. Windows version available at course home page.

Gada68k, cross-compiler for MC68340, only Linux

For lab assignment, programming with the train simulator.

A simple resource handler (Lab Related…)

Assume a single resource is shared.

Clients of the resource call "Acquire" to access it

Clients of the resource call "Release" to free it

When multiple clients want to access it, only one client get access; others are bolcked.


```
procedure P1 is
   protected type resource is
        entry Acquire;
        procedure Release;
   end resource;
    ----Body of the protected object
   r1: Resource;
   task  client1;
   task  client2;
   task body client1 is
        r1.Acquire; …  r1.release;
   end client1;
   task body client2 is
        r1.Acquire; …  r1.release;
   end client2;
begin
        null
end P1;
```

```
procedure P1 is
   protected type resource is
        entry Acquire;
        procedure Release;
   Private
        free: boolean:=true;
   end resource;
   protected body resource is
      entry Acquire when free is
      begin
        free:=false;
      end;
      procedure release is
      begin
        free:=true;
      end;
  end resource;
…  …  …
end P1;
```

Assignment 8

Three integer variables is shared by several tasks. Write an ADA package **Notice_Board** containg ***read*** and ***write*** operations by concurrent tasks, for these variables. The following type is declared:

```
type var_num is range 1 .. 3;
```

The package shall include the following procedures:

```
procedure read (num : in var_num; value : out integer)
        -- Returns value of variable denoted by 'var_num'.
        -- Block the calling task if the variable

        -- not have been previously assigned through 'write'.

procedure write (num : in var_num; value : in integer)
        -- Assign 'value' to variable denoted by 'var_num'.
```

Write operations are mutual exclusive for a particular variable, i.e writing one variable should not block operations on another variable. Hint: Create a protected object for each variable).

Step 1, the specification

```
package Notice_Board is
   type Var_Num is range 1 .. 3;
   procedure read( Num : in Var_Num; Value : out Integer);
   procedure write( Num : in Var_Num; Value : in Integer);
end Notice_Board;
```

Goes to specification file, e.g. "`Notice_Board.ads`"

Declarations are visible throughout the application

Step 2, declarations

Goes to declaration file, e.g. "`Notice_Board.adb`"
Details the implementation, also contains locals i.e. visible inside but not outside the package and "privates", i.e. unique copies for every object instance.

```
package body Notice_Board is
   protected type Protected_Int is
      entry Read( Value : out Integer);
      procedure Write( Value : in Integer);
   private
      X : Integer := 0;
      Written : Boolean := False;
   end Protected_Int;

   protected body Protected_Int is
   -- implementation of entry 'Read' and local procedure 'Write' (protected)
   end Protected_Int;

   -- multiple instances of the protected object...
   type Protected_Int_List is array (Var_Num) of Protected_Int;
   Board_Variables : Protected_Int_List;
 -- Exported (visible) procedures
   procedure read( Num : in Var_Num; Value : out Integer) is
   ... -- implementation of procedure 'Read' globally visible
   procedure write ( Num : in Var_Num; Value : in Integer) is
   ... -- implementation of procedure 'Write' globally visible
end Notice_Board;
```


Step 3, protected details

```
protected body Protected_Int is

    entry Read(Value : out Integer) when Written is
    begin
       Value := X;
    end;

    procedure Write(Value : in Integer) is
    begin
       X := Value;
       Written :=  True;
    end;

 end Protected_Int;
```

Note that '`Value`' is unique for every instance of the `Protected_Int` object.

The choice of an *entry* for `Read` is motivated by the required guard (`Written`).

Step 4, globally visible procedure details

```
procedure Read( Num : in Var_Num; Value : out Integer) is
begin
    Board_Variables(Num).Read(Value);
end;

procedure Write ( Num : in Var_Num; Value : in Integer) is
begin
    Board_Variables(Num).Write(Value);
end;
```

'Num' indicates the actual instance of the protected object

Recommended home work...

Eloborate on the following assignments..

1-5

Will get you started and going with the IDE and ada95 taking mechanisms.

6

Learn how to make a set of procedures "generic" simply by using *types*.

7

A simple exercise on protected objects.

10

Preparations for the laboratory assignments.

Summary

- Types and program flow control
- Compilation units, "packages"
- Tasks in Ada95
- Protected objects in Ada95
- High value hints for persuing this course

ADA Distilled

Course home page provides links to all needed resources.

Lab Description, Exercise Compendium, Ada Distilled…