Ada Pointers – Access type

Access type

- Pointers in Ada are called access type
- They point to objects located at certain addresses
- Objects of access types are implicitly initialized with null - pointing to nothing when not explicitly initialized
- used rarely in Ada, there are other ways without pointers

```
type Person is record
First_Name : String (1..30);
Last_Name : String (1..20);
end record;
type Person_Access is access Person;
```

Access type

```
Father: Person_Access := new Person;
-- uninitialized
Mother: Person_Access := new
Person'(Mothers_First_Name, Mothers_Last_Name);
-- initialized
```

access the object in the storage pool by appending .all

`Mother.all` is the complete record;
components are denoted as usual with the dot notation

```
Mother.all.First_Name
```

When accessing components, *implicit dereferencing* (i.e. omitting .all) is more convenient

```
Mother.all := (Last_Name => Father.Last_Name,
First_Name => Mother.First_Name); -- marriage
```

Access type

```
type Vector is array (1 .. 3) of
Complex;
type Vector_Access is access Vector;
VA: Vector_Access := new Vector;
VB: array (1 .. 3) of Vector_Access :=
(others => new Vector);
C1: Complex := VA (3); -- a shorter
equivalent for VA .all (3)
C2: Complex := VB (3)(1); -- a shorter
equivalent for VB(3).all (1)
```

Access type

- deep and shallow copies when copying with access objects:

```
Obj1.all := Obj2.all; -- Deep copy:
Obj1 still refers to an object
different from Obj2, but it has the
same content

Obj1 := Obj2; -- Shallow copy: Obj1
now refers to the same object as Obj2
```

Access type

- Ada standard mentions a garbage collector, which would automatically remove all unneeded objects that have been created on the heap

```
with Ada.Unchecked_Deallocation;
procedure Deallocation_Sample is
type Vector is array (Integer range <>) of Float; type
Vector_Ref is access Vector;
procedure Free_Vector is new Ada.Unchecked_Deallocation
(Object => Vector, Name => Vector_Ref);
VA, VB: Vector_Ref;
 V : Vector;
begin
VA := new Vector (1 .. 10); VB := VA; -- points to the
same location as VA
VA.all := (others => 0.0); -- ...
Free_Vector (VA); -- The memory is deallocated and VA is
now null
V := VB.all; -- VB is not null, access to a dangling
pointer is erroneous
end Deallocation_Sample;
```

Access type

- When the keyword **all** is used in their definition, they grant read-write access

```
type Day_Of_Month is range 1 .. 31; type
Day_Of_Month_Access is access all
Day_Of_Month;
```

- General access types granting read-only access to the referenced object use the keyword constant
- The referenced object may be a constant or a variable.

```
type Day_Of_Month is range 1 .. 31; type
Day_Of_Month_Access is access constant
Day_Of_Month;
```

Access type

```
type General_Pointer is access all Integer;
type Constant_Pointer is access constant Integer;
I1: aliased constant Integer := 10; I2: aliased
Integer;
P1: General_Pointer := I1'Access; -- illegal
P2: Constant_Pointer := I1'Access; -- OK, read only
P3: General_Pointer := I2'Access; -- OK, read and
write
P4: Constant_Pointer := I2'Access; -- OK, read only
P5: constant General_Pointer := I2'Access; -- read
and write only to I2
```

Access type

- *Anonymous access types* are in two versions
- general access types, granting either read-write access or read-only access depending on whether the keyword constant appears
- anonymous access can be used as a parameter to a subprogram or as a discriminant

```
procedure Modify (Some_Day: access
Day_Of_Month);
procedure Test (Some_Day: access constant
Day_Of_Month);
type Day_Data (Store_For_Day: access
Day_Of_Month) is record -- components
end record;
```

Access type

- An access to subprogram allows the caller to call a subprogram without knowing its name nor its declaration location

```
type Callback_Procedure is access procedure
(Id : Integer; Text: String);
type Callback_Function is access function
(The_Alarm: Alarm) return Natural;


procedure Process_Event (Id : Integer; Text:
String);
My_Callback: Callback_Procedure :=
Process_Event'Access;
```

Ada - Protected

Protected

- allows encapsulation and safe usage of shared variables without the need for any explicit mutual exclusion and signaling mechanisms
- tasks that safely manage locally-declared data with regard to multiple clients
- **protected** modules, based on the well-known computer science concept of *monitor*
- a protected module encapsulates a data structure and exports subprograms that operate on it under automatic mutual exclusion
- it also provides automatic, implicit signaling of conditions between client tasks

Protected

- a protected module can be either a single protected object or a protected type, allowing many protected objects to be created.
- a protected module can export only procedures, functions and entries
- its body may contain only the bodies of procedures, functions and entries
- the protected data is declared after **private** in its specification, but is accessible only within the protected module's body

Protected

- protected procedures and entries may read and/or write its encapsulated data, and automatically exclude each other
- protected functions may only read the encapsulated data
- multiple protected function calls can be concurrently executed in the same protected object, with complete safety
- but protected procedure calls and entry calls exclude protected function calls, and vice versa
- exported entries and subprograms of a protected object are executed by its calling task

Protected

- like a task entry, a protected entry can use a guard to control admission
- this provides automatic signaling, and ensures that when a protected entry call is accepted, its guard condition is True, so that a guard provides a reliable precondition for the entry body

```
protected type Protected_Buffer_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
private Buffer : Item;
        Empty : Boolean := True;
end Protected_Buffer_Type;
```

Protected

```
protected body Protected_Buffer_Type is
entry Insert (An_Item : in Item) when
Empty is
begin Buffer := An_Item;
Empty := False; end Insert;
entry Remove (An_Item : out Item) when
not Empty is
begin An_Item := Buffer; Empty := True;
end Remove;
end Protected_Buffer_Type;
```

Protected

- the state variable Empty ensures that messages are alternately inserted and removed,
- no attempt can be made to take data from an empty buffer
- this is achieved without explicit signaling or mutual exclusion constructs, whether in the calling task or in the protected type itself
- calling a protected entry or procedure is exactly the same as that for calling a task entry
- easy to replace one implementation of the abstract type by the other, the calling code being unaffected

Protected

```
task type Semaphore_Task_Type is
entry Initialize (N : in Natural);
entry Wait;
entry Signal;
end Semaphore_Task_Type; ...

task body Semaphore_Task_Type is
Count : Natural;
begin
accept Initialize (N : in Natural) do Count := N; end Initialize;
loop
select
when Count > 0 => accept Wait do Count := Count - 1; end Wait;
or
accept Signal; Count := Count + 1;
end select;
end loop;
end Semaphore_Task_Type;
```

Protected

The task can be used like:

Full, Free : Semaphore_Task_Type; …

Full.Initialize (0); Free.Initialize (nr_Slots); …

Free.Wait; Full.Signal;

- Initialize and Signal operations of this protected type are unconditional, so they are implemented as protected procedures
- the Wait operation must be guarded and is therefore implemented as an entry.

Reimplemented with protected:

Protected

```
protected type Semaphore_Protected_Type is
procedure Initialize (N : in Natural);
entry Wait;
procedure Signal;
private Count : Natural := 0;
end Semaphore_Protected_Type;
```

- Unlike the task type above, this does not ensure that Initialize is called before Wait or Signal, and Count is given a default initial value instead

Protected

```
protected body Semaphore_Protected_Type is
procedure Initialize (N : in Natural) is
begin Count := N;
end Initialize;
entry Wait when Count > 0 is
begin Count := Count - 1; end Wait;
procedure Signal is
begin Count := Count + 1; end Signal;
end Semaphore_Protected_Type;
```

Exception

- `Tasking_Error`, when a task cannot be activated because the operating system has not enough resources, e.g.
- when calling a terminated task it raises `Tasking_Error` in the caller

Attributes

- X'Callable is an Ada attribute where X is any task object. If the task is completed or has been terminated, this attribute is false. Otherwise, this attribute is true (i.e. the task is callable).
- calling X'Callable can result in a race condition. X'Callable may be true at the time the attribute value is read, but it may become false at the time action is taken based on the value read. Once X'Callable is false, however, it can be expected to stay false.

Attributes

- X'Terminated is an Ada attribute where X is any task object. This attribute indicates if X has terminated (true) or not (false).
- calling X'Terminated can result in a race condition. X'Terminated may be false at the time the attribute value is read, but it may become true at the time action is taken based on the value read. Once X'Terminated is true, however, it can be expected to stay true.

Ada – select statement

Select - termination

- tasks often contain infinite loops to allow them to serve an arbitrary number of calls in succession
- control cannot leave a task's parent until its tasks terminate, need a way to know when it should terminate
- it is done by a *terminate alternative* in a selective wait

The task terminates when:

- at least one terminate alternative is open, and
- there are no pending calls to its entries, and
- all other tasks of the same parent are in the same state (or already terminated), and
- the task's parent has completed (i.e. has run out of statements to execute).

- First two conditions ensure that the task can stop, it is ready for it.
- The other two conditions ensure that the task has no negative effects on other parts as there are no further calls that might change its state

Select - terminate

```
task type Terminating_Buffer_Task_Type is
    entry Insert (An_Item : in  Item);
    entry Remove (An_Item : out Item);
 end Terminating_Buffer_Task_Type;
 ...
 task body Terminating_Buffer_Task_Type
is
    Datum : Item;
 begin
    loop
      select
        accept Insert (An_Item : in  Item) do
          Datum := An_Item;
        end Insert;
      or
        terminate;
      end select;
      select
        accept Remove (An_Item : out Item)
        do
          An_Item := Datum;
        end Remove;
      or
        terminate;
      end select;
    end loop;
end Terminating_Buffer_Task_Type;
```

Select - delay

- Relative delay: delay Duration_Expression;
- Absolute delay: delay until Time_Expression;
- Duration is a special Float type, Time is provided by Ada.Calendar
- delay alternative in a selective wait statement allows a task to no more accept calls after a maximum delay in achieving rendezvous with any client

```
task Resource_Lender is
    entry Get_Loan (Period : in Duration);
    entry Give_Back;
 end Resource_Lender;
 ...
 task body Resource_Lender is
    Period_Of_Loan : Duration;
 begin
    loop
  ......
   end loop;
 end Resource_Lender;
```

Select - delay

```
select
    accept Get_Loan (Period : in Duration) do
          Period_Of_Loan := Period;
     end Get_Loan;
       select
            accept Give_Back;
            or
            delay Period_Of_Loan;
            Log_Error ("Borrower did not give up loan soon enough.");
       end select;
  or
     terminate;
end select;
```

Select - until

```
task body Current_Position is
begin  ...
  loop
    select
      accept Request_New_Coordinates (X :   out Integer;  Y :   out Integer) do
        -- copy coordinates to parameters   ...
      end Request_New_Coordinates;
    or
      delay until Time_To_Execute;
    end select;

    Time_To_Execute := Time_To_Execute + Period;
    -- Read Sensors and execute coordinate transformations
  end loop; ...
end Current_Position;
```

Time_To_Execute is of type Ada.Calendar.Time and can get value by Ada.Calendar.Clock which provides system time

Select - else

- else alternative in a selective wait statement allows a task to not accept calls where the rendezvous can not be achieved immediately
- the receiver task wants immediate rendezvous, which means the caller has to be executing the entry point call statement in the very same moment, or the call has to be already in the called entry's queue
- the task is not waiting for communication in this case, continues with the else part
- if any delay might appear, then the **or delay** version has to be used

Select - else

```
task body Buffer_Messages is   ...
begin ...
  loop
    delay until Time_To_Execute;
    select
      accept Get_New_Message (Message : in     String) do
        -- copy message to parameters ...
      end Get_New_Message;
    else  -- don't wait for rendezvous
      -- perform built in test Functions  ...
    end select;
    Time_To_Execute := Time_To_Execute + Period;
  end loop;   ..
end Buffer_Messages;
```

Select – timeout

- A *timed entry call* lets a client specify a maximum delay before achieving rendezvous, failing which the attempted entry call is withdrawn and an alternative sequence of statements is executed.
- To time out the *functionality* provided by a task, two distinct entries are needed: one to pass in arguments, and one to collect the result. Timing out on rendezvous with the latter achieves the desired effect.

Select – timed call or delay

- Timed entry calls
- An entry call can be made waiting a well defined time interval, so that it is withdrawn if the rendezvous is not achieved after the defined time. This uses the select statement notation with an **or delay** part, the construct is:

```
select
   Callee.Rendezvous;
 or
   delay 1.0;
 end select;
```

Select – timed call or delay

```
task Password_Server is
   entry Check (User, Pass : in String; Valid : out Boolean);
   entry Set (User, Pass : in  String);
end Password_Server;
...
User_Name, Password : String (1 .. 8);
...
Put ("Please give your new password:");
Get_Line (Password);
select
   Password_Server.Set (User_Name, Password);
   Put_Line ("Done");
or
   delay 10.0;
   Put_Line ("The system is busy now, please try again later.");
end select;
```

Select - else

- Conditional entry calls
- An entry call can be made conditional, so that it is withdrawn if the rendezvous is not immediately achieved. This uses the select statement notation with an **else** part, the construct is:

```
select
   Callee.Rendezvous;
 else
   Do_something_else;
 end select;
```

Select – or delay 0.0

- Conditional entry calls
- the attempt to start the rendezvous may take some time, especially if the callee is on another processor, so the *delay 0.0;* may expire although the callee would be able to accept the rendezvous, whereas the *else* construct is safe.
- Seems equivalent to or delay 0.0 (busy waiting)

```
select
   Callee.Rendezvous;
 or
   delay 0.0;
   Do_something_else;
 end select;
```

Parallel Programming Tasks

Tasks

- A *task unit* is a program unit that is running concurrently with the main program and other tasks of an Ada program
- is called a *task* in Ada terminology, and is similar to a *thread, e.g. in Java*
- the execution of the main program is also a task, the anonymous environment task (parent task)
- a task unit has both a declaration and a body (mandatory) a task body may be compiled separately as a subunit, but a task may not be a library unit, nor a generic.
- every task depends on a *master (or parent)*, which is the surrounding declarative region - a block, a subprogram, another task, or a package - that declared the task

Tasks

- The execution of a master does not complete until all its dependent tasks have terminated.
- The environment task is the master of all other tasks; it terminates only when all other tasks have terminated.
- Task units are similar to packages in that a task declaration defines entities exported from the task, whereas its body contains local declarations and statements of the task.

Tasks

```
task Action is
declarations of exported identifiers
end Action; ...
task body Action is local
declarations and statements
end Action;
```

A task declaration can be simplified, if nothing is exported:

```
task No_Exports_Actions;
```

Tasks

```
procedure Housework is
task Clean;
task Cook;
task body Clean is ... end Clean;
task body Cook is ... end Cook;
-- the two tasks are automatically
created and begin their execution
begin -- Housework
null; -- Housekeeping waits here for
them to terminate
end Housework;
```

Tasks

- It is possible to declare task types, allowing task units to be created dynamically, and placed in data structures

```
task type TaskTypeName is
...
end TaskTypeName; ...
Task_1, Task_2 : TaskTypeName;
...
task body TaskTypeName is
...
end TaskTypeName;
```

- Task types are **limited**, i.e. they are restricted in the same way as limited private types, so assignment and comparison are not allowed

Tasks

- **Rendezvous** – communication via entry points
- The only entities that a task may export are entries
  An **entry** looks like a procedure. It has a name identifier and may have **in**, **out** or **in out** parameters.
- Communication from task to task: by the *entry calls*
- Information passes between tasks through the actual parameters of the entry call
- Tasks encapsulate data structures within and operate on them by entry calls, in a way analogous to the use of packages for encapsulating variables

Tasks

- an entry is executed by the called task, not the calling task, which is suspended until the call completes
- if the called task is not ready to service a call on an entry, the calling task waits in a (FIFO) queue associated with the entry
- the interaction between calling task and called task is known as a *rendezvous*
- the calling task requests rendezvous with a specific named task by calling one of its entries
- a task accepts rendezvous with any caller of a specific entry by executing an **accept** statement for the entry.
- If no caller is waiting, it is held up
- the entry call and accept statement behave symmetrically

Tasks – buffer example

```
task type Buffer_Task_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
end Buffer_Task_Type; ...
Buffer_Pool: array (0..10) of Buffer_Task_Type;
Item1 : Item; ...
task body Buffer_Task_Type is
Datum : Item;
begin
loop
accept Insert (An_Item : in Item) do Datum := An_Item;
end Insert;
accept Remove (An_Item : out Item) do An_Item := Datum;
end Remove;
end loop;
end Buffer_Task_Type; ...
Buffer_Pool(1).Remove (Item1);
Buffer_Pool(2).Insert (Item1);
```

Tasks – selective wait

- To avoid being held up when it could be doing productive work, a task often needs the freedom to accept a call on any one of a number of alternative entries
- the *selective wait* statement allows a task to wait for a call on any of two or more entries
- if only one of the alternatives in a selective wait statement has a pending entry call, then that one is accepted
- if two or more alternatives have calls pending, the implementation is free to accept any one of them, it chooses one at random, introduces *bounded non-determinism* into the program

Tasks

```
task type Variable_Task_Type is

entry Store (An_Item : in Item);
entry Fetch (An_Item : out Item);

end Variable_Task_Type;

task body Variable_Task_Type is
Datum : Item;
begin
… *
end Variable_Task_Type;

x, y : Variable_Task_Type; -- x, y two tasks
```

Tasks

```
accept Store (An_Item : in Item) do
Datum := An_Item;
end Store;
loop
select
accept Store (An_Item : in Item) do
Datum := An_Item;
end Store;
or
accept Fetch (An_Item : out Item) do An_Item :=
Datum;
end Fetch;
end select;
end loop;
```

Tasks

```
item1 : Item; ...
x.Store(An_Expression); ...
x.Fetch (item1); y.Store (item1);
```

- a task of Variable_Task_Type must be given an initial value by a first Store operation before any Fetch operation can be accepted
- the acceptance of any alternative can be conditional by a *guard*, which is *Boolean* precondition for acceptance
- it makes easy to write monitor-like tasks
- no need for an explicit signaling mechanism, nor for mutual exclusion
- an alternative with a True guard is said to be *open*
- it is an error if no alternative is open when the selective wait statement is executed, and raises Program_Error exception

Tasks

```
task Cyclic_Buffer_Task_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
end Cyclic_Buffer_Task_Type; ...
task body Cyclic_Buffer_Task_Type is Q_Size : constant := 100;
subtype Q_Range is Positive range 1 .. Q_Size; Length : Natural
range 0 .. Q_Size := 0;
Head, Tail : Q_Range := 1;
Data : array (Q_Range) of Item;
begin  .....
end Cyclic_Buffer_Task_Type;
```

Tasks

```
select when Length < Q_Size =>
accept Insert (An_Item : in Item) do
Data(Tail) := An_Item;
end Insert;
Tail := Tail mod Q_Size + 1;
Length := Length + 1;
or
when Length > 0 =>
accept Remove (An_Item : out Item) do
An_Item := Data(Head);
end Remove;
Head := Head mod Q_Size + 1; Length :=
Length - 1;
end select;
```