Ada Exercises – Sequential and Object-Oriented Programming

The goal of this lab session is to exercise various aspects of the Ada Language, both imperative and object oriented features.

If you feel you need additional information, you may consult `http://en.wikibooks.org/wiki/Ada_Programming`

Using code for this lab session

All source code is provided in the archive `ada_lab.tar.gz`. To open this archive, issue the following command "`tar zxvf ada_lab.tar.gz`" this will create the directory `ada_lab` with all source code.

A makefile is provided to compile all source code. Issue "`make help`" for more details.

Assignments

You are tasked to perform all exercises, and write an implementation report. This implementation report is made of two files : a text file describing what you did, and the annotated source code.

Hard deadline : one week after the day of the lab session.

1 A small introduction to Ada

Exercise 1 (Introduction to GCC) *In this exercise, we study the compilation process using the* GNU Compiler Collection*, or* `gcc`*, and the* GNAT *Ada front-end.*

We consider the canonical "Hello World!" program, its source code is :

```
with Ada.Text_IO;

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

In the following, we suppose the source code of the "Hello World!" example is in file `hello.adb`*. To create this file, simply open a text editor (emacs, vi, gedit, ...) and insert the source code.*

To compile it, we issue the following command :

```
gnatmake hello.adb
```

Finally, to run the program, open a terminal, and type, in the directory where you compiled the binary :

```
neraka-2:c hugues$ ./hello
Hello World!
```

– *Compile and execute this program.*

Exercise 2 (Float manipulation) *In this exercise, we review float manipulation and computation.*

- *Write two functions to compute the volume and surface of a sphere. We recall that* $V = \frac{4}{3}\pi r^3$ *and* $S = \pi r^2$.

  The package `Ada.Numerics` *defines the* `Pi` *constant whose value is an approximation of* π *for the target.*
- *Write a function* `Compute_Pi` *that evaluates the value of* π *using the development of the arctan function.*

  We recall that $\pi = 4\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$.

  The function has one parameter which is the number of iterations and returns a float. Compare with the value of `Ada.Numerics.Pi`.
- *Computing digits of Pi can be made generic, the generic part being the actual type returns (a* `Float`*, a Long_Flaot, . . .).*

  Write a function `Compute_Pi_Generic` *whose formal generic parameter is any floating point type. Its signature in Ada is given by*

```
generic
   type T is digits <>;
function Compute_Pi_Generic (Iter : Natural) return T;
```

  To use this function, you have to instantiate it, e.g. define a function with T being defined :

```
function My_Compute_Pi is new Compute_Pi_Generic (Float);
```

Exercise 3 (Bubble sort) *In this exercise, we want to implement the "bubble sort" algorithm using generics.*

The bubble sort is a very simple algorithm to sort an array, it can be described in pseudo code as follow :

Algorithm 1 Bubble Sort

```
procedure SORT(Arr : T_Array)
   repeat
      switched ← false
      for j over all indexes of T, but last do
         if index "J" of T is less than index "J+1" of T then
            swap elements at positions J and J+1
            switched ← true
         end if
      end for
   until switched = false
end procedure
```

Basically, bubble sorts iterate as often as required over the array. When two consecutive elements are unsorted, it swaps them. The function stops when no more elements are unsorted. The term "bubble" comes from the analogy with bubbles in a glass of champagne.

This algorithm takes as parameter an unsorted array, and returns it sorted. We want to implement it using Ada generics. We note that the actual types of the elements in the array is irrelevant. All we need is a total order relationships of these elements.

We use this information to provide a simple interface to this algorithm. We will provide three formal parameters :

- *the actual type of the elements, T;*
- *a comparison function, defining a total order on elements of T;*
- *a print function, for debugging purposes.*

The specification of the bubble sort is given as follows :

```
generic
   type T is private;

   with function Less (X : T; Y : T) return Boolean;
   --  Return true iff X is less than Y

   with function Print (X : T) return String;
   --  Return image of X

package Bubble is

   type T_Array is array (Integer range <>) of T;

   procedure Sort (Arr : in out T_Array);
   --  Sort elements of Arr

   procedure Print_Array (Arr : T_Array);
   --  Print elements of Arr

end Bubble;
```

– *Implement this specification and test it carefully. Place test code in a file called* `test_bubble.adb`.

Note : use Ada attributes (`Range`, `First`, `Last`, ...*) to support flexibility in your design. Actually, Ada arrays may start at positions different from* 0 *or* 1.

Exercise 4 (Smart pointers) *The misuse of pointers is a major source of bugs : the constant allocation, deallocation and referencing that must be performed by a program written using pointers introduces the risk that memory leaks will occur.*

A "smart pointer" is an abstract data type that simulates a pointer while providing additional features, such as automatic garbage collection. Smart pointers try to prevent memory leaks by making the resource deallocation automatic : when the pointer (or the last in a series of pointers) to an object is destroyed, for example because it goes out of scope, the referenced object is destroyed too.

In this exercise, we will implement smart pointers using `Controlled` *types. Ada controlled types are tagged types (objects) with constructor (*`Initialize`*), destructor (*`Finalize`*) and assignments (*`Adjust`*) primitives. Implementing reference counting smart pointers can be implemented using these three primitives. See Ada RM §7.6 for more details on controlled types.*

The following proposes a specification for a generic smart pointer package. The `Encapsulated` *type defines the entity pointed to by the pointer,* `Ref` *is the actual smart pointer. It is a tagged type, inheriting primitives from* `Ada.Finalization.Controlled`*. The accessors* `Get` *and* `Set` *are used to change the values of the smart pointers. The private primitives* `Adjust` *and* `Finalize` *handle the reference counting mechanisms :*

– *in case of assignment,* `Adjust` *is called, incrementing the internal counter ;*
– *in case a variable of type* `Ref` *is finalized, as a result of exiting its scope,* `Finalize` *is called, decrementing the counter.*

We rely on default record initialization to properly set all data when creating a variable of type `Ref`*. Besides, we rely internally on pointers to share counters and pointed date. Great care must be taken to ensure there is no dangling references !*

– *Propose an implementation for this package, and the corresponding tests. Place test code in a file called* `test_smart_pointers.adb`.
Use debug traces to show the various behaviors.

```
--  This package provides support for reference counting.
--
--  A Smart_Pointer plays the role of an access type (although it is
--  not an access type), and keeps a reference to the designated
--  entity. When a smart pointer goes out of scope, the designated
--  entity's reference count is automatically decremented.
--
--  When the reference count reaches 0, the corresponding entity is freed.

with Ada.Finalization;

generic
   type Encapsulated is private;
package Smart_Pointers is
   type Encapsulated_Access is access all Encapsulated;

   type Ref is tagged private; --  The smart pointer type
   Null_Ref : constant Ref;    --  Null reference

   procedure Set (Self : in out Ref; Data : Encapsulated);
   procedure Set (Self : in out Ref; Data : access Encapsulated);
   --  Replace the current contents of Self.
   --  Data is adopted by the smart pointer, and should no longer be
   --  referenced directly elsewhere. The reference count of Data is
   --  incremented by 1.
   --  Typical code looks like:
   --      Tmp := new Encapsulated;
   --      Set (Ptr, Tmp);
   --  (You can't do
   --      Set (Ptr, new Encapsulated);
   --   for visibility reasons)

   function Get (P : Ref) return Encapsulated_Access;
   pragma Inline (Get);
   --  Return a pointer the data pointed to by P.
   --  We return an access type for efficiency reasons. However, the
   --  returned value must not be freed by the caller.

   overriding function "=" (P1, P2 : Ref) return Boolean;
   --  Whether the two pointers point to the same data

   function Get_Refcount (Self : Ref) return Natural;
   --  Return the current reference count.
   --  This is mostly intended for debug purposes.

private
   type Refcounted is record
      Data : Encapsulated_Access;
      Counter : Integer := 0;
   end record;
   type Refcounted_Access is access all Refcounted;

   type Ref is new Ada.Finalization.Controlled with record
      Entity : Refcounted_Access;
   end record;
```

```
   overriding procedure Finalize (P : in out Ref);
   overriding procedure Adjust   (P : in out Ref);
   --  Take care of reference counting

   Null_Ref : constant Ref :=
     (Ada.Finalization.Controlled with Entity => null);

end Smart_Pointers;
```