Data Structures Lab 3 - Containers Fall 2019 Burris

Due: Wednesday October 30 for MWF and Thursday October 31 for TTH prior to the start of class:

Submission: For all options the results should appear first followed by the complete code!

Most modern programming languages offer a "container" class for the convenience of users. Containers are typically inefficient with respect to CPU time and memory allocation compared to static types. They may however offer great convenience for the developer. Commonly offered container classes include one for constrained objects and a second for unconstrained objects. These classes typically are further refined to incorporate containers termed "sequence" and "associative." Sequence classes hold sequences of related items. Associative classes associate a key with each element of the class then manipulate elements based on the keys. You may not utilize container classes in any language for any lab during the semester unless the lab specifically states you must use the container class. Ada provides the following container for "defined/constrained" classes:

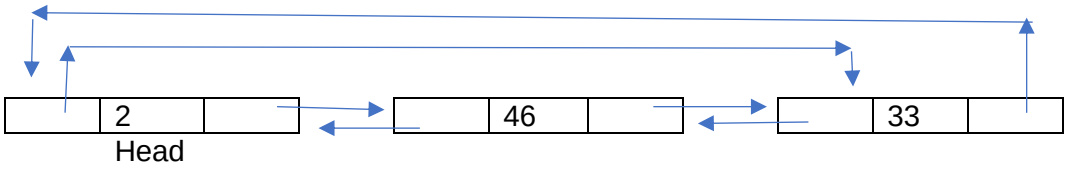
a) Ada.Containers.Vectors
b) Ada.Containers.Doubly_lLinked_Lists
c) Ada.Containers.Hashed_Maps
d) Ada.Containers.Ordered_Maps
e) Ada.Containers.Hashed_Sets
f) Ada.Containers.Ordered_Sets

A similar group of classes/packages is provided for undefined/unconstrained types using similar names including Ada.Containers.Indefinite_Vectors with specialized generic procedures such as Ada.Containers.Generic_Array_Sort and Ada.Containers.Generic_Constrained_Sort. The purpose of this lab is to develop the basic technology to implement (program) these capabilities in languages including Ada, C++, Java, Python and Smalltalk using more basic constructs. Implementation of containers, complex numbers, etcetera using code placed in libraries extends the capabilities of languages. This technique is widely used to extend programming language capabilities and convenience for modern languages.

"C" Option: Homogeneous (maximum grade is 75):

Hint: appendix 1, 2 and 3. For the "C" Option remove the generic in appendix 2, replace "item" with Integer and set max to a specific value like 20. Appendix 3 is closest to the lab. Appendix 6 is helpful if you need to pass functions or operator overloads.

Implement package/class Integer_Containers allowing the user to store a list of integers whose meaning is application dependent with respect to the user (a homogeneous list). You must store elements in a doubly linked list (container) with a list head. The following represents a list with 33 inserted first followed by 46 on the right side (front) of the head node. The list head should contain the number of integers currently stored in the list.



A container (package/class) definition would normally contain procedures and/or functions to insert at the head of the list, insert at the rear of the list, provide the number of items currently in the list, search for an item returning a pointer to it if found (null if not found), print the contents of the item given a pointer to the node containing the item and ask for the next item in the list traversing the list from the front to rear. Finally, you must be able to delete a random item from the doubly linked list given its location (pointer to the item). A partial package/class specification might include the following:

```
package Integer_Containers is
        type Integer_Container is private;
        type Integer_Node_Point is access Integer_Container;
        procedure InsertFront( list: in out Integer_Container, amt: in Integer, Success: Boolean);
        procedure InsertRear( list: in out Integer_Container, amt: in Integer, success: out Boolean);
        function listSize( list: in Integer_Container) return Integer;
        function findItem( list: in Integer_Container, aValue: Integer) return Integer_Node_point);
        procedure delete(list: in out: Integer_Container, Integer_NodePoint);
        -- Additional functions and procedures as needed.
private
        type Integer_Container is record
                value: Integer;
                next, previous: Integer_Pointer;
        end Integer_Container;
  -- rest of specification.
end Integer_Containers;  -- followed by the body in another file

--Main program
        MyList: Integer_Container;  -- create an integer container as a doubly linked list.
```

Process all "C" option transactions in the order specified.

a) Insert 33 in front (right).
b) Insert 57 in front
c) Insert 85 at the rear (left).
d) Insert 62 at the rear.
e) Insert 95 at the front.
f) Print the contents of the list from front to rear.
g) Print the content of the list from rear to front.
h) Find and delete the node containing 57.
i) Find and delete the node containing 33.
j) Find and delete the node containing 33.

k) Find and delete the node containing 62.
l) Insert 22 in front.
m) Delete the node containing 95.
n) Print the contents of the list from front to rear.

"B" Option – Homogeneous (maximum grade is 80):

Hint: Appendix 1,2, 3. Appendix 6 is helpful if you need to pass functions or operator overloads.

You need not do the "C" option. Write a generic/template package/class List_Package allowing the user the ability to store any programmer defined type in a doubly linked list with the operations defined in the "C" option. Use a list head. You will probably wish to provide an overload for the "=" operator as a generic parameter. Implement the package/class List_Package allowing for any user defined in a doubly linked list. Each entry in the list other than the head node may be used to store a user defined transaction. Data fields in the head node may be used or ignored by the implementer. The basic package/class definition will contain at least the following methods:

```
generic
        type ItemsType is private;
package List_Package is
        -- Methods for previous grading option
        --   See sample specification/code for similar application below inCompStacg.
end List_Package – followed by the body in another file

-- main program
with List_Package;
procedure MainLine is
        type ItemType is ( Shoes, Kites, Jacks, Food);
        currentItem: ItemType;
        price: Float;
        amt: Integer;

        type InventoryItem is record
                itemName: ItemType; unitPrice: Float; inStock: Integer;
        end InventoryItem;

        temp, theItem: InventoryItem;

package InventoryList is new List_Package (InventoryItem); use InventoryList;
```

Process the following transactions in the specified order after creating homogeneous containers for cars and planes (two separate lists). You may use the code for cars and planes used in the examples below if desired. Place the cars and planes in the correct lists.

a) Insert a Ford with 4 doors at the rear.
b) Insert a Ford with 2 doors at the front.
c) Insert a GMC with 2 doors at the rear.
d) Insert a RAM with 2 doors at the rear.
e) Insert a Chevy with 3 doors at the front.
f) Print the number of items in the list.
g) Print the contents of the list (front to rear).
h) Find and delete the first Ford in the list (search front to rear).
i) Print the number of items in the list.
j) Print the contents of the list (front to rear).
k) Insert a plane with 3 doors and 6 engines by Boeing at the front.
l) Insert a plane with 2 doors and 1 engine by Piper at the front.
m) Insert a plane with 4 doors and 4 engines by Cessna at the front.
n) Print the list.

Hint: You do not need the "abstract stack (inheritance from a common ancestor) to create the car and plane types in appendix 5. You may use my code for cars and planes in your code.

"A" Option: Heterogeneous Container using Inheritance (maximum grade is 90). Appendix 6 is helpful if you need to pass functions or operator overloads.

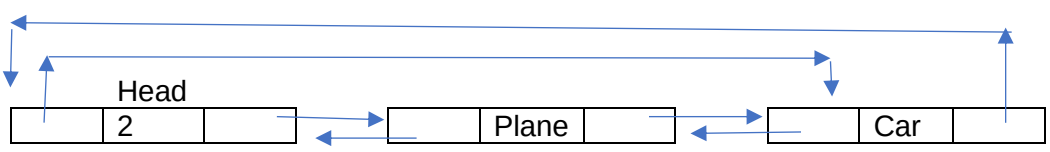
Hint: Appendix 1,4 and 5 for all "A" Options.

You need not do the "C" or "B" options. Rather implement a single package/class using inheritance to allow multiple data types/objects in a heterogeneous list.

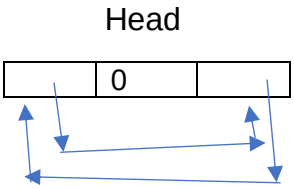
First: Use your package to process all "C" option transactions! You may have to create a new list element to store integers in your "A" option list.

Second: Use this package/class to implement a single doubly linked list and process all "B" Option transactions placing the cars and planes in a single list using inheritance.

A sample list with one plane and one car follows:



An example of the empty list follows:



"A+" Option: Heterogeneous Container using Inheritance (maximum grade is 95):

Complete the "A" option. Allow at least one **task** to place items in the list and a second task to remove items from the list.

"A Super +" Option: Heterogeneous Container using Inheritance (maximum grade is 100):

Complete the "A" option. Allow multiple tasks to place items in the list and multiple tasks to remove items from the list preventing RACE conditions!

Appendix 1: For all options.

In most industries employee records (inventory records etcetera) appear 20 or more times in applications. Rather than define them in every application it is convenient do define them once then utilize the same definition for the abstraction in every application. Not only is this convenient but increases sharing, consistency across applications, and ease of update for all applications, The following class illustrates the sharing concept for dates.

```
-- in file CompStk1.ads
-- Exports IntIO, MonthName, MonthNmaeIO, Date and PrintDate.
with Ada.Text_IO; use Ada.Text_IO;
package CompStk1 is
  package IntIO is new Ada.Text_IO.Integer_IO(Integer);
  use IntIO;

  type MonthName is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
                     Oct, Nov, Dec);  -- Enumeration type.

  package MonthNameIO is new – Ada generates i/O routines.
            Ada.Text_IO.Enumeration_IO(MonthName);
  use MonthNameIO;
```

```
  type Date is record
    Month: MonthName;
    Day: Integer range 1..31;  -- Limits range.
    Year: Integer;             -- No limit on range.
  end record;

  procedure PrintDate(aDate: Date);
end CompStk1;

--in file CompStk1.adb
package body CompStk1 is

 procedure PrintDate(aDate: Date) is
  begin
    put("mmm/dd/yyyy is "); put(aDate.Month);put("/");
    put(aDate.Day,2); put("/"); put(aDate.Year,4);
  end PrintDate;
end CompStk1;
```

Appendix 2: For "C" Option

Sample **homogeneous** stack for intrinsic data types and programmer data type including task types.

```
generic                    -- in file Gstack.ads
  max:integer;             -- size of stack
  type item is private;    -- type to stack
package gstack is
        procedure push(x: in item);
        procedure pop(x: out item);
end gstack;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

package body gstack is    -- in file Gstack.adb
        s:array(1..max) of item;  -- allocate in stack.
        top: integer range 0..max;

        procedure push(x: in item) is
        begin
           top := top + 1;   s(top) := x;
        end push;

        procedure pop(x: out item ) is
```

Sample run:

```
enter an integer:  10

enter an integer:  20

enter an integer:  30

enter an integer:  40

result of pop  40

result of pop  30
```

```
        begin
           x := s(top);    top := top - 1;
        end pop;

begin
        top := 0; --initialize top of stack to empty
end gstack;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


with Ada.Text_IO; use Ada.Text_IO;  -- in file Gusestac.adb
with gstack;   -- generic stack defined in gstack10.ads /.adb
procedure genstack is
   package IIO is new Ada.Text_IO.Integer_IO(integer); use IIO;

        package integer_stack is new gstack(100,integer);
        use integer_stack;
        m: integer;
begin
        for i in 1..4 loop
                put("enter an integer ");    get(m);    push(m);
        end loop;

        for i in 1..4 loop
                put("result of pop ");    pop(m);    put(m);    new_line;
        end loop;
end genstack;

-- Consider adding:  package intStk is new gstack(10,integer); use intStk;
-- Now push(m) will result in a compile time error as "Push" cannot be
--uniquely determined from the context.  Use intStk.push(m) or
--integer_stack.push(m), the full object oriented notation.
```

Now consider using the "date" type. A skeleton follows.

```
with Ada.Text_IO; use Ada.Text_IO;
with gstack;;
with CompStkg;  use CompStkg;

procedure UCmpStkg is  -- in file UCmpStkg
        stackSize: integer := 15;
  package DateStack is new gstack(stackSize, Date);
  use DateStack;
-- rest followed by access, shown 2 ways
        push(aDate);  --if stack is clearly identifiable (unique0 or to specify using
        DateStack.push( aDate );   -- DateStac if there are multiple date stacks.
```

Appendix 3: Stack implemented as a linked list.

The following creates a **homogeneous** stack using a linked list. The "C" option does not require the generic as each stack only stores a specific predefined type. The generic allows for the user to store any intrinsic or user defined type (including tasks, protected types, etc.). Replacing "MyType with "integer," Float," Character, Date etcetera creates the desired stack type.

```
generic   -- in file CompStkg.ads
  type MyType is private;
package CompStkg is

  procedure Push(X: MyType);
  function  Pop return MyType;  -- Note parenthesis are not required if there are no parameters.

private

  type Node;  -- Avoid recursive definition.
  type NodePtr is access Node;  -- Define pointer type to Node.

  type Node is record  -- Allocated in heap at run time.
    MyData: MyType;
    Next: NodePtr;
  end record;
end CompStkg;

with Ada.Unchecked_Deallocation;  -- in file CompStkg.adb
package body CompStkg is
-- Provide opportunity for garbage collection and reuse of Nodes.
  function free is new Ada.Unchecked_Deallocation(Node, NodePtr);  -- reclaim heap storage.
  Head, pt: NodePtr := null;  -- Ada actually sets all pointers to null when they are declared.

  procedure Push(X: MyType) is
  begin                    -- No check for overflow.
    Head := new Node'(X, Head);  -- Allocated from returned memory, heap is none returned.
  end Push;  -- Most languages return Head = null if out of storage.
```

```
 function  Pop return MyType is
   X: MyType;
 begin
   X := Head.MyData;         -- No check for underflow.
    pt := Head;
   Head := Head.Next;
   free(pt);   -- Memory hemorraging occurs if forgotten.
   return X;
 end Pop;

end CompStkg;

--  Example of programming by "Composition" (bottom-up, creating whole from parts)
--  as opposed to programming by "Classification"
--  (top-down) better known as the use of inheritance.

with Ada.Text_IO; use Ada.Text_IO;
with CompStk1; use CompStk1;
with CompStkg;
procedure UCmpStkg is  -- in file UCmpStkg

  package CharStack is new CompStkg(character);
  use CharStack;

  package DateStack is new CompStkg(CompStk1.Date);
  use DateStack;

  Char: Character;
  ADate: Date;

begin
  Push('A');  Push('B');  Push('C');
  put(pop);   put(Pop);   put(Pop);  new_line(2);

  Push((Jan, 15, 1992)); Push((Mar, 24, 1994));
  Push((Jun, 12, 1999));
  ADate := Pop;  PrintDate(ADate); new_line;
  Adate := Pop;  PrintDate(Adate); new_line;
  Adate := Pop;  PrintDate(ADate); new_line;
end;
```

Sample Output:
CBA

mmm/dd/yyyy is JUN/12/1999
mmm/dd/yyyy is MAR/24/1994
mmm/dd/yyyy is JAN/15/1992

Appendix 4: "B/A" Option to Implement Containers.

The preceding examples are limited to stacks containing a single data type, i.e., "**homogeneous**" lists. Industrial grade applications may find it necessary to store more than one type/class item in the container. The assumption is the "**heterogeneous**" items will have similar operations on the data. The software will select the appropriate version of the method for the user selected operation, i.e., polymorphism. Ada is polymorphic by default. We start with an example of homogeneous inheritance (tagged types) then expand the example to allow heterogeneous data types. The secret in most object oriented languages (Java, C++, SamllTalk, Ada, etc.) is for all children to be derived from a common parent type. Similar operations (methods, procedures, functions) on the data type employ polymorphic methods to implement the behavior for each desired type. The container for the parent type implies the ability to include children derived from the parent.

```
-- Creation of a "cube" type from a "rectangle" type using inheritance.
--in file taged1.adb
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure taged1 is

 type Rectangle is tagged  -- tagged allows inheritance
  record
    length:   Integer;
    width:    Integer;
  end record;

 function Size(r: in Rectangle) return Integer is  -- "intrinsic" method
  begin   return r.length * r.width;   end Size;

 -- create a cube by inheriting from Rectangle.

 type Cube is new Rectangle with – create cube using inheritance
       record     height:   Integer;    end record;

 -- Cube inherits fields length, width, and function Size.
 -- This size may be redefined as:
```

```
  function Size(c: Cube) return Integer is  -- intrinsic function
    begin
      return Size( Rectangle( C ) ) * C.height;  -- cast “Cube” as a rectangle.
    end Size;

  -- Note the type conversion "Rectangle(c)" so that the inherited
  -- function Size for Rectangle (overload) can be used.

  rect1:  Rectangle := (6,10);
  cube1:  Cube := (length => 6, width => 10, height => 20);
begin
    put( Size( rect1 ) );      put( Size( cube1 ) );
    rect1 := Rectangle( cube1 );   cube1 := ( rect1 with 20 );
end taged1;

-- User-written subprograms are classified as primitive operations
-- if they are declared in the same package specification as the
-- type and have the type as a parameter or result. Derived types
-- inherit all primitive operations that belong to the parent type.
```

Appendix 5: “B/A” Heterogenous Abstract Type taking advantage of inheritance. **Ada allows definition/implementation of related types/classes in the same file.**

This example uses the object notation for methods as follows to creating intrinsic methods for a package:
“procedures:” <method-name> (<object>, <parameters>)
“functions::” <function-name>(<object>, <parameters>)
return <return-type>

```
-- Programing by  “Classification” (top-down) as opposed to
-- composition by “Composition” (bottom-up).

--  In file AbstStck.ads,  Creation of abstract stack.
package AbstStck is
  type AbstractStack is limited private;

  type AbstractStackElement is tagged private;
  type AbstractStackElementPtr is
        access all AbstractStackElement'Class;
        --Allows access to AbstractStackElement and any class inheriting
        --(created from using inheritance) from AbstractStackElement.
```

```
  procedure Push(Stack: access AbstractStack; Y: in AbstractStackElementPtr);
  function   Pop(Stack: access AbstractStack) return AbstractStackElementPtr;
  function  StackSize(Stack: AbstractStack) return integer;
-- function “=” (aNode: AbstractStackElementPtr,
                valueToCompare: genreric-parameter-for-comparison) return Bollean;

private
  type AbstractStackElement is tagged – Allow for heterogeneous
    record                                    -- stacks via inheritance.
      Next: AbstractStackElementPtr;
    end record;

  type AbstractStack is limited
    record
      Count: integer := 0;  -- used to track the number of items in stack.
      Top: AbstractStackElementPtr := null;
    end record;
end AbstStck;

-- In file AbstStck.adb
package body AbstStck is
procedure Push(Stack: access AbstractStack; Y: in AbstractStackElementPtr) is
    Pt: AbstractStackElementPtr;
  begin
    Y.Next := Stack.Top;  Stack.Top := Y;  Stack.Count := Stack.Count + 1;
  end Push;

  function Pop(Stack: access AbstractStack) return AbstractStackElementPtr is
   Pt: AbstractStackElementPtr;
  begin
   if Stack.Top = null then -- Check for underflow.
     return null;
   end if;
   Stack.Count := Stack.Count - 1;
   Pt := Stack.Top;  Stack.Top := Stack.Top.Next;  -- Pop stack, note hemmoraging.
   return Pt;    -- Storage should be returned to an available storage list for applications
  end Pop;     -- with high activity or executing for extended periods of time.

  function StackSize(Stack: AbstractStack) return integer is
  begin  return Stack.Count;  end StackSize;
end AbstStck;
```

Creation/use of a Car type used with a stack.

```
-- in file MakeCar.ads
with AbstStck;
package MakeCar is

  type String4 is new String(1..4);

  type Car is new AbstStck.AbstractStackElement with record
     NumDoors:  integer;
     Manufacturer: String4 := "GMC ";  -- Sample default value.
  end record;

  procedure AssignNumDoors(aCar: in out Car; N: in integer);

  procedure AssignManufacturer(aCar: in out Car; Manu: in String4);

  procedure PrintNumDoors(aCar: in Car);

  procedure PrintManufacturer(aCar: in Car);

  procedure IdentifyVehicle(aCar: in Car);
end MakeCar;

-- in file MakeCar.adb
with Ada.Text_IO; use Ada.Text_io;
with AbstStck;
package body MakeCar is
  package IntIO is new Ada.Text_IO.Integer_IO(Integer);  use IntIO;

  procedure AssignNumDoors(aCar: in out Car; N: in integer) is
  begin aCar.NumDoors := N; end AssignNumDoors;

  procedure AssignManufacturer(aCar: in out Car; Manu: in String4) is
  begin aCar.Manufacturer := Manu; end AssignManufacturer;

  procedure PrintNumDoors(aCar: in Car) is
  begin put("Num doors = "); put(aCar.NumDoors); new_line; end PrintNumDoors;

  procedure PrintString4(PrtStr: String4) is
  begin for I in 1.. 4 loop
     put(PrtStr(I));
  end loop; end PrintString4;
```

```
  procedure PrintManufacturer(aCar: in Car) is
  begin put("Manufacturer is "); PrintString4(aCar.Manufacturer); new_line; end;

  procedure IdentifyVehicle(aCar: in Car) is
  begin
    put("Car with "); put(aCar.NumDoors, 4); put(" doors");
    put(" made by "); PrintString4(aCar.Manufacturer); new_line;
  end IdentifyVehicle;
end MakeCar;

--in file UAbstStc.adb: place cars in the stack.
with Ada.Text_IO; use Ada.Text_io;
with AbstStck; use AbstStck;
with MakeCar; use MakeCar;

procedure UAbstStc is
  type Stack_Ptr is access AbstractStack;
  CarStack:  Stack_Ptr := new AbstractStack;

  StackPoint: Stack_Ptr;
  NewCar, CarPt: AbstractStackElementPtr;

begin  --Create 1st car.
  NewCar := new Car'(AbstractStackElement with 4, "Ford"); --Heap allocation
  push(CarStack, NewCar);

  NewCar := new Car;  -- Create 2nd car.
  AssignNumDoors(Car'Class(NewCar.All), 2);
  AssignManufacturer(Car'Class(NewCar.all), "Chev");
  push(CarStack, NewCar);

  NewCar := new Car;  -- Create 3rd car.
  AssignNumDoors(Car'Class(NewCar.All), 2);
  -- Default manufacturer to "GMC ".
  push(CarStack, NewCar);

  for I in 1..StackSize(CarStack.all) loop
    CarPt := pop(CarStack);
    PrintManufacturer(Car'Class(CarPt.All));
    PrintNumDoors(Car'Class(CarPt.All));
    new_line;
```

```
  end loop;
end UAbstStc;
```

Sample Output:

```
Manufacturer is GMC
Num doors =         2

Manufacturer is Chev
Num doors =         2

Manufacturer is Ford
Num doors =         4
```

```
-- file MakePlane.ads:  Create planes for use with heterogeneous container.
with AbstStck;
package MakePlane is
  type String8 is new String(1..8);

  type Plane is new AbstStck.AbstractStackElement with record
     NumDoors:  integer;
     NumEngines:  integer;
     Manufacturer: String8 := "Boeing  ";
  end record;

  procedure AssignNumDoors(aPlane: in out Plane; N: in integer);

  procedure AssignManufacturer(aPlane: in out Plane; Manu: in String8);

  procedure AssignNumEngines(aPlane: in out Plane; NE: in integer);

  procedure PrintPlane(aPlane: in Plane);

  procedure IdentifyVehicle(aPlane: in Plane);
end MakePlane;

-- In file MakePlane.adb
with Ada.Text_IO; use Ada.Text_io;  with AbstStck;
package body MakePlane is
  package IntIO is new Ada.Text_IO.Integer_IO(Integer);  use IntIO;

  procedure AssignNumDoors(aPlane: in out Plane; N: in integer) is
  begin aPlane.NumDoors := N; end AssignNumDoors;

  procedure AssignManufacturer(aPlane: in out Plane; Manu: in String8) is
```

```
  begin aPlane.Manufacturer := Manu; end AssignManufacturer;

  procedure AssignNumEngines(aPlane: in out Plane; NE: in integer) is
  begin aPlane.NumEngines := NE; end AssignNumEngines;

  procedure PrintString8(PrtStr: String8) is
  begin for I in 1..8 loop  put(PrtStr(I));   end loop; end PrintString8;

  procedure PrintPlane(aPlane: in Plane) is
  begin
      put("Num doors for plane = "); put(aPlane.NumDoors, 4); new_line;
      put("Number engines = "); put(aPlane.NumEngines); new_line;
      put("Manufacturer = "); PrintString8(aPlane.Manufacturer); new_line;
  end PrintPlane;

  procedure IdentifyVehicle(aPlane: in Plane) is
  begin
    put("Plane with "); put(aPlane.NumDoors, 4); put(" doors, ");
    put(aPlane.NumEngines, 4); put(" engines, made by ");
    PrintString8(aPlane.Manufacturer); new_line;
  end IdentifyVehicle;
end MakePlane;
```

Using heterogeneous stack with cars and planes in same stack.

```
with Ada.Text_IO; use Ada.Text_io;
with AbstStck; use AbstStck;
with MakeCar, MakePlane; use MakeCar, MakePlane;

procedure  UAbstSt2 is
  type Stack_Ptr is access AbstractStack;
  VehicleStack:  Stack_Ptr := new AbstractStack;

  StackPoint: Stack_Ptr;
  NewCar, CarPt, NewPlane, PlanePt, VehiclePt:
         AbstractStackElementPtr;

begin
  NewCar := new Car'(AbstractStackElement with 4, "Ford");  -- Heap allocation!
  push(VehicleStack, NewCar); -- 1st car.

  NewPlane := new Plane'(AbstractStackElement with 2, 2, "Northrup");  -- in heap!
```

```
  push(VehicleStack, NewPlane);  --1st plane.

for I in 1..StackSize(VehicleStack.all) loop
    VehiclePt := pop(VehicleStack);
    if VehiclePt.all in Car then  -- ** Identify class of object at run time.
      IdentifyVehicle(Car'Class(VehiclePt.all));
    elsif VehiclePt.all in Plane then
      IdentifyVehicle(Plane'Class(VehiclePt.all));
    end if;
    new_line;
  end loop;

end UAbstSt2;
```

Sample Output:

```
Plane with    2 doors,    2 engines, made by Northrup

Car with    4 doors made by Ford
```

****** Heterogeneous versus Homogeneous!**

Appendix 6: Passing Functions and Operator Overloads

```
-- In file GIOEX.ads. Creates a rectangle with user defined data type for length and
-- width. Note that it is frequently desirable to pass methods including I/O routines
-- for programmer defined data types.
-- The following demonstrates how to pass I/O procedures to a generic package.
-- The rectangle may also be used for inheritance if desired.

generic
   type MyType is private;
   with function "*"(X,Y: MyType) return MyType;
   with procedure Put(X: MyType);
package GIOEX is
  type Rectangle is tagged
    record
      Length:   MyType;
      Width:    MyType;
    end record;
  function Size(r: in Rectangle) return MyType;  -- intrinsic functions
  function RectLength(r: in Rectangle) return MyType;
end GIOEX;

--in file GIOEX.adb
  with Ada.Text_IO; use Ada.Text_IO;  -- Access restricted to body.
package body GIOEX is

  function Size(r: in Rectangle) return MyType is
    begin
      put("The Size of the Rectangle with length "); put(r.Length);
      put(" and width "); put(r.Width);  put(" is ");
      put( r.Length * r.Width);  put("!"); new_line(2);
      return r.Length * r.Width;
    end Size;

  function RectLength(r: in Rectangle) return MyType is
  begin
    put("The length is "); put(r.Length);
```

```
   new_line;
   return r.Length;
  end;
end GIOEX;

-- Sample program to show how to pass a programmer defined data type
--  and I/O methods to a generic.
-- in file UGIOEX.adb

with GIOEX;
with Ada.Text_IO; -- Use Ada.Text_IO;
procedure UGIOEX is

  package MyFloatIO is new Ada.Text_IO.Float_IO(Float);
  use MyFloatIO;

--The generic put statement format in Ada.Text_IO.Float_IO.

--  procedure Put( item: float;
--              fore: Ada.Text_IO.field := 0;     -- "0" means use minimum space.
--              aft:  Ada.Text_IO.Field := 0;
--              exp:  Ada.Text_IO.Field := 0
-- );

--  Supply an overload for the generic written by the compiler in MyFloat_IO.
  procedure MyPut(X: Float) is
  begin   MyFloatIO.Put(X, 0, 0, 0);  end;

-- Note that "*" is defined for the intrinsic type Float.
  package MyGIOEX is new GIOEX( Float, MyPut, "*");
  use MyGIOEX;

  Rect1: Rectangle := (5.0, 6.0);  //Create class object using constructor!
  Len: Float;

begin
```

```
  Len := Size(Rect1);  Len := RectLength(Rect1);
end UGIOEX;
```

```
c:\>UGIOEX
The Size of the Rectangle with length 5.0 and width 6.0 is
30.0!    -- 3.00000E+01

The length is 5.0    --  5.00000E+00
```

```
-- in file usegioex2.adb.  Use of Venus units for measurement.
-- These operations are required for our human mission to Venus.
```

Sample overloading the "*" Operator

```
with Ada.Text_IO; use Ada.Text_io;
with GIOEX;

procedure UseGioex2 is
   package MyIntIO is new Ada.Text_IO.Integer_IO(Integer);
   package MyFloatIO is new Ada.Text_IO.Float_IO(Float);

   type VenusMeasure is record
      F1: Integer;
      F2: Float;
   end record;

  -- Define I/O for VenusMeasurement.
  procedure Put(v: VenusMeasure) is
  begin
    MyIntIO.put(v.F1);
    put(" ");
    MyFloatIO.put(v.F2);
    new_line;
  end;

  -- Define mutliplication for VenusMeasurement.
  function "*"(p1: VenusMeasure; p2: VenusMeasure)
        return VenusMeasure is
    temp: VenusMeasure;
  begin
    temp.F1 := p1.F1 * p2.F1;
    temp.F2 := p1.F2 * p2.F2;
    return temp;
```

```
   end;

   package MyVenusRectangle
         is new GIOEX(VenusMeasure, Put, "*");
   use MyVenusRectangle;

   width: VenusMeasure := ( 5, 5.5);
   height: VenusMeasure := (3, 2.4);

   Rect1: Rectangle := (width, height);  -- Creates a rectangle using
                                         -- Venus measurements.
   Ans: VenusMeasure;

begin
   Ans := Size( Rect1 );
end UseGIOEx2;
```