Data Structures Dynamic Lists Spring 2021 Burris

Due: Friday April 9, 2021 prior to the start of class:

Submission: Provide a cover page for your lab containing at least your name, the date of the submission, and the grading option completed. For all options, the results must appear first followed by the complete code! Your code should contain comments including your name, statement of what the lab accomplishes and competition date. A lab may not be submitted more than 4 times! Data for all lab options is in DynList.txt. To receive maximum credit on each grading option, all stack and heap space must be reclaimed prior to allocating space for the next grading option! You may delete lines in DynList.txt not required for your program, e.g., comments.

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 native types. They may however offer great convenience for the developer. Commonly offered container classes include one for constrained objects (homogeneous) and a second for unconstrained objects (heterogeneous). Most container classes are further refined to incorporate containers termed “sequence” and “associative.” Sequence classes hold sequences (lists) of related items. Associative classes (dictionaries) 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 may use containers. Ada provides the following containers for “defined/constrained” classes:

1. Ada.Containers.Vectors
2. Ada.Containers.Doubly\_lLinked\_Lists
3. Ada.Containers.Hashed\_Maps
4. Ada.Containers.Ordered\_Maps
5. Ada.Containers.Hashed\_Sets
6. 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 types of capabilities in languages including Ada, C++, Java, Python, Groovy/Grails, Ruby/Rails 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.

*The above containers may be of great use. They are however generic in the sense they try to be all things for all application areas and developers. Professional software engineers/programmers should expect to be tasked with creating mission specific software/containers with the above characteristics but tailored to individual business needs with extensions. This lab will prepare us for that demand.* ***Please do the “A” option***. You will need the technology from the “A” option in future labs to meet grading standards. You may use Ada, C++ or Java for all labs this semester. This course will emphasize compiled languages for systems/problems with “hard real time constraints.” In these systems hardware efficiency is typically more important than programmer convenience/efficiency. An example would be software used to land a passenger jet during less than ideal conditions. While you may use Java for all labs, Java will typically limit your lab grade to a “B” as it will not be able to meet all problem specifications. Ada and C++ are compiled languages. Java is interpretive. Interpretive languages frequently execute 10 to 50 times slower than compiled solutions for the same problem. You will receive extensive programming help for labs in Ada. Minimal help will be provided C++ and Java.

**“C” Option: Homogeneous (maximum grade is 75):**

Dear Ima Programmer,

NASA and its primary contractors have been very enthusiastic with respect to our previous “linear list mission specific software” for the trip to Mars. As a result, we now have a contract for a similar product using dynamic storage allocation. If you do not feel you are up to the task, please update your resume. You will be terminated and provided the opportunity to travel, meet new people and explore new opportunities at your own expense. Our initial design requirements with syntax/semantic examples follow. We will extend these requirements in future labs. **YOU MUST CODE SEQUENTIAL TEXT FILES FOR ALL I/O OPERATIONS**. Appendix 2 contains sample code to read and write sequential text files. Sample data structure as envisioned by NASA follows for 3 departments used for a personnel application. Exact details for the contents of each node will be provided later in this document. Only process the “C” option transactions in the file DynList.txt.

|  |  |  |
| --- | --- | --- |
|  | Department  List Head | Departments |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Betty  ID  Payrate |  |  | Tom  ID  Payrate |  |  | Sara  ID  Payrate |  | | link  num = 3 | Accounting |
|  | link 🡨Ω  num = 0 | IT |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Adam  ID  Payrate |  |  | Bob  ID  Payrate |  |  | Sue  ID  Payrate |  | | link  num = 3 | Sales |

A program meeting partial specifications created using AdaGuide appears in Appendix 1. The above structure was selected to sort random transactions by department. **Departments** must be an enumeration type. NASA requires fast base plus displacement access to departments. Circular list have been selected for the dynamic list nodes for their advantages with respect to operations such as concatenation and efficient return of storage to the system. The link structure was selected as we have no information as to the number of transactions for a given department during a mission but can estimate on upper bound on the require space required. The array of list heads must be allocated in the stack at run time (box1.adb). List nodes should be allocated dynamically in the heap when needed and returned to available storage when deleted for reuse.

**The requirements specify dynamically allocated circular list** (Data Structures Class Notes, Link Allocated Circular List, approximately pages 67-69) for tracking inventory. Sample list declarations and functionality follow:

**“C” Option methods:**

|  |  |  |
| --- | --- | --- |
| insertNodeLeft( deptNameIn: DepartmentType; empNameIn: NameType; titleIn: TitleType; IDin: Integer; Payratein:Float) | New left most node. Increment number in list. | IL |
| insertNodeRight( deptNameIn: DepartmentType; empNameIn: NameType; titleIn: TitleType; IDin: Integer; Payratein:Float ) | New right most node. Increment number in list. | IR |
| deleteNodeLeft( deptNameIn) | Delete left most node. Decrement number in list. | DL |
| deleteNodeRight( deptNameIn: DepartmentType) | Delete right most node. Decrement number in list. | DR |
| deleteDepartment(deptNameIn: DepartmentType) | Efficiently delete all nodes in the department list. Number in list = 0. | DD |
| printAllDepartments( ) | Print each department name and all nodes neatly left to right. | PA |
| printDepartment(deptNameIn: DepartmentType) | Print neatly all nodes for the specified department. | PD |

The instruction code on the right in the chart should be implemented as an enumeration type. We have been encouraged to use them as labels in ”CASE/SWITCH” statements.

**Sample transactions (actual data in file LabDynamicListsSp2021Data.txt):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Code** | **Department** | **Name/Vendor** | **Title** | **ID** | **Payrate** |
| IL | Sales | Mary | Sales\_person | 1378 | 25.46 |
| IL | Sales | Mary | Sales\_person | 1364 | 20.13 |
| IR | Sales | Vallerie | Manager | 1358 | 27.15 |
| IL | Media | Marty | Manager | 1444 | 25.17 |
| IR | Media | Bob | spokesperson | 1416 | 10.13 |
| IL | Crew | Sally | Pilot | 675 | 35.11 |
| IR | Media | David | Designer | 1412 | 20.76 |
| DL | Media |  |  |  |  |
| DR | Media |  |  |  |  |
| PA |  |  |  |  |  |
| PD | Media |  |  |  |  |
| DD | Sales |  |  |  |  |
| PD | Sales |  |  |  |  |
| \*\* | Additional | transactions | For “B” Option | follow! |  |
| IL | Sales | Chevy | SUV | 7862 |  |
| IL | Crew | Ford | Raptor | 7914 |  |
| IR | Crew | GMC | Pickup | 7777 |  |
| IR | Crew | Dodge | Ram | 7647 |  |
| IR | Crew | Chevy | StingRay | 7813 |  |
| IL | Crew | Joe | Copilot | 643 | 45.62 |
| DR | Media |  |  |  |  |
| DL | Sales |  |  |  |  |
| DR | Media |  |  |  |  |
| PA | Crew |  |  |  |  |
| PD | Media |  |  |  |  |
| && | Additional | Transactions | for “A” Option. |  |  |
| DD | Crew |  |  |  |  |
| IS | Crew | Sally | Pilot | 7001 | 46.00 |
| IS | Crew | Bob | missionSpecialist | 7643 | 41.56 |
| IS | Crew | Marty | Copilot | 7032 | 45.55 |
| IS | Crew | Freedom | Orbiter | 8763 |  |
| IS | Crew | Atlantis | Orbiter | 8731 |  |
| IS | Crew | Challenger | Orbiter | 8777 |  |
| IS | Crew | Chevy | SUV | 7633 |  |
| IS | Crew | Northrup | Jet | 6387 |  |
| PD | Crew |  |  |  |  |
|  |  |  |  |  |  |

The instruction codes in the right column should be implemented as an enumeration type. We have been encouraged to use them as labels in ”CASE/SWITCH” statements. You should have a list head for all departments enumerated by

**Department types include “type DepartmentType is (Sales, Crew, IT, Media, Accounting, noDept);**

**Job types for all grading options include “type JobType is (Programmer, Software\_Engineer, Sales, Inventory\_Control, customer, manager);**

**Name include type NameType is (Bob, Mary, Sally, David, Marty, Vallerie, Sam, Joe, noName); You are free to utilize names as an enumeration type or as character strings.**

**For your convenience, the test data is store in**

**“B” Option: Heterogeneous (maximum grade is 85):**

Dear Ima Programmer,

You need not explicitly code the “C” Option. Rather, use it as a guide for our new expanded contract to associate employees, planes and cars with departments to which they are assigned. The solution must be extensible allowing new software to utilize any new programmer defined types in lists without the need to modify existing programs. Some software may need to be recompiled to take advantage of the new capabilities. You must utilize inheritance, polymorphism and function/operator overloading to meet NASA’s specification. Passing user defined print routines to classes/packages is desirable. We will continue the **use of static storage in the stack for the array of department list heads** indexed using an enumerations type to minimize the time to access department list entries. Dynamic storage allocated in the heap must be used for items in department **circular lists** as we will not know ahead of time how many employees or pieces of equipment will be assigned to the departments. Circular list will continue to be used for the advantages they provide over traditional singly linked lists. Process all “C” followed by all “B’ transactions. Process all “C” option followed by all “B” option transactions in the file DynList.txt.

**“B” and “A” Option Design Specification:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node |  | Node |  | Node |  |  | Array of List Heads | Array Index |
|  |  |  |  |  |  |  |  |  |
| AbstractList  Pointer |  | AbstractList  Pointer |  | AbstractList  Pointer |  |  | Abstract List Pointer | Accounting |
|  |  |  |  |  |  |  |  |  |
| car |  | Plane |  | employee |  |  |  |  |
| data structures |  | data structures |  | Data structures |  |  |  |  |
| methods |  | methods |  | methods |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| Abstract List Pointer |  | Abstract List Pointer |  | Abstract List Pointer |  |  | Abstract List Pointer | Sales |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| Car |  | Plane |  | Employee |  |  |  |  |
| 2 door  ChevyID |  | 2 door  2 engine  Northrup  ID |  | Mary  SalesPerson  ID |  |  |  |  |
| PrintAll(--)  etcetera |  | PrintAll(--)  etcetera |  | PrintAll(--)  etcetera |  |  |  |  |

*Hints: Dynamic storage allocation at runtime for sequentially allocated structures using generics ( control of data type, values, etcetera) in Ada is explained on pages 52-59 of “Data Structures Program Class Notes” Sample code.“ “box1.ada,” page 58 of the Program Class Notes exhibits an example for dynamically allocating storage for arrays/lists in the stack at runtime for the “C” option.*

**“A” Option: Heterogeneous (maximum grade is 95):**

Dear Ima Programmer,

NASA requires list creation using any programmer defined data structures accessed via programmer defined enumeration types. Please consider converting the “B” option code to generics (Ada) or templates (C++/Java). Create list via generic instantiation. Use of generics is encouraged but not required! You need not explicitly write the “C” and “B” option code. Rather use your “A” option code to create all lists for your program using the same physical stack space for the “C,” “B” and “A” option lists/arrays. **AGAIN, YOU MAY NOT USE I/O REDIRECTION. All data/transactions must be read from a sequential text file and all results printed to a sequential text file!** **You must use programmer defined enumeration types effectively.**

|  |  |  |
| --- | --- | --- |
| insertSorted( deptNameIn: DepartmentType; NameIn: NameType; titleIn: TitleType; IDin: Integer; Payratein:Float) | Insert transactions in lexicographic order by ID. Increment number in list. | IS |

See the data file for typical transactions.

Management expects to view the “C” option results followed by the “B” option results and finally the “A” Option results.

**“A+” Option: Heterogeneous (maximum grade is 105):**

Take the final step to professional grade. Make the “A” option generic with respect to department types. Declare the stack list heads inside the generic package so the user just says insert left or right, delete left or right etcetera and just passes the record/object to be manipulated. Hide the inheritance mechanism from the user population if possible**. The “>” operator used in lexicographic insertion could be passed as a generic parameter**. Package AbstSck in the programming notes (approximately page 65) is very close to what is required. An example of using the generic package appears in package MakeCar and UAbstStc” on pages 66-68 of the programming notes.

**On the cover page, add a few notes to tell me what you have done to accomplish the “A+” option requirements.**

**You are encouraged but not required to use generic/template packages/classes for the “A” Option. You must use generics for the “A+” option.**

The following provides is a sample of what transactions should look like. This is not code, just a logical example using partial objects (just department):

type JobType is (Programmer, Software\_Engineer, Sales, Inventory\_Control, customer, manager);

pt, length: Integer;

Package AList is new MakeList( jobList, 20, Job\_Type, etcetera); -- generics (box1.ada)

PrintList( ); -- [ null ]

insertSorted( Programmer );

insertSorted ( Software\_Engineer);

insertSorted ( Software\_Engineer);

insertSorted ( Sales);

insertSorted ( customer);

insertSorted ( Programmer);

PrintList( ); -- [ Programmer, Programmer Software\_Engineer, Software\_Engineer,

-- Sales, customer] *I am not concerned with upper/lower case output at pesent*.

Delete( 4 );

PrintList( ); -- Programmer, Programmer Software\_Engineer, Sales, customer]

-- Print all information for each entry allowing for easy reading by the user.

**Appendix 0**

CODING HINTS FOR ALL GRADING OPTIONS, emphasizing the “C” Option:

generic -- in file CreateList.ads

size: integer;

type itemType is private;

package CreateList is -- export the following behavior.

procedure AddToList( listItem: in itemType );

procedure PrintList; -- Ada does allows but not require empty ( ).

procedure PrintList( pt: in integer );

procedure DeleteList( pt: in integer );

function ListLength return integer; -- Ada does allows but not require empty ( ).

End CreateList;

package body CreateList is -- in file CreateList.adb

len: Integer;

list: array(1..size) of itemType;

procedure AddToList( ListItem: in itemType ) is begin null; end AddToList;

procedure PrintList is begin null; end PrintList;

procedure PrintList( pt: in Integer) is begin null; end PrintList;

procedure DeleteList( pt: in Integer) is begin null; end DeleteList;

function ListLength return Integer is

begin

return len;

end ListLength;

end CreateList;

**-- Generic close to “A+” option.**

**--I did not take advantage of the generic for the “C” option example below.**

-- Lab 1 in fle Lab1.adb

with Ada.Text\_IO; use Ada.Text\_IO;

with CreateList;

procedure Lab1 is -- in file Lab1.adb

-- declartions

-- package IIO is new Ada.Text\_IO.Integer\_IO(Integer); use IIO;

size: Integer;

begin

put("enter size for 'C' option: ");

get(size); -- for “C” Option

declare

-- for "C" option declare list directly as in box1.adb.: array

Clist: Array(1..Size) of Integer; **-- Allocate in stack;**

begin

-- process transactions for "C" option using a loop to read and print results

null;

end;

**-- CList no longer in scope. Space popped/reclaimed automatically from stac**

get(size); -- for "B" option

declare

-- use the same format for creating the "C" list if you skip the "C" lab option.

--package BList is new CreateList( 3, float ); use BList; if using generics for “B” option.

-- static package structures/variables allocated in the stack.

Blist: Array(1..Size) of Integer; **-- Allocate in stack;**

floatNum: float;

begin

-- process transactions for "B" option.

--loop --for or while statement

AddToList(floatNum); -- or full object oriented notation

BList.AddToList(floatNum);

-- end loop

end;

**-- BList no longer in scope. Space popped/reclaimed automatically from stack.**

get(size); -- for "A" option

declare

type JobType is (Programmer, Software\_Engineer, Sales, Inventory\_Control, customer, manager);

package JobTypeIO is new Ada.Text\_IO.Enumeration\_IO(JobType); use JobTypeIO;

-- use the same format for creating the "C" and “B” list if you skip the "C" lab option.

-- package AList is new CreateList( 20, JobType ); use AList; Allocate in stack.

--Alist: Array(1..Size) of Integer; **-- Allocate in stack; if do not use generics.**

job: JobType;

begin

-- process transactions for "A" option.

-- loop -- for or while statement

AddToList( job ); -- or with full object oriented notation

AList.AddToList( job );

-- end loop

end;

**-- AList no longer in scope. Space popped/reclaimed automatically from stack.**

end Lab1;

**Appendix 1:**

with Ada.Text\_IO; Use Ada.Text\_IO; -- Created with AdaGuide.

procedure L2Doption is

package IIO is new Ada.Text\_IO.Integer\_IO(Integer); use IIO;

type DepartmentType is (Sales, Crew, IT, Media, Accounting, noDept);

package DeptTypeIO is new Ada.Text\_IO.Enumeration\_IO(DepartmentType); use DeptTypeIO;

type TitleType is (manager, Sales\_person, Designer, Programmer, Software\_Engineer,

spokesPerson, Pilot, Copilot, scientist, missionSpecialist, noTitle );

package TitleTypeIO is new Ada.Text\_IO.Enumeration\_IO(TitleType); use TitleTypeIO;

type NameType is (Bob, Mary, Sally, David, Marty, Vallerie, Sam, Joe, noName);

package NameTypeIO is new Ada.Text\_IO.Enumeration\_IO(NameType); use NameTypeIO;

type EmpNode;

type EmpPt is access EmpNode;

type EmpNode is record

deptName: DepartmentType := noDept;

empName: NameType := noName;

title: TitleType := noTitle;

next: EmpPt := null;

end record;

type DepartmentNode;

type DeptPt is access DepartmentNode;

type DepartmentNode is record

deptName: DepartmentType;

empName: NameType;

num: Integer := 0;

next: EmpPT := null;

end record;

Department: array( DepartmentType ) of DepartmentNode;

procedure insertNodeLeft( deptNameIn: DepartmentType; empNameIn: NameType; titleIn: TitleType ) is

pt: EmpPt;

begin

pt := new EmpNode'(deptNameIn, empNameIn, titleIn, null);

if( Department(deptNameIn).next = Null ) then -- empty list

pt.next := pt;

Department(deptNameIn).next := pt;

else -- list not empty

pt.next := Department(deptNameIn).next;

Department(deptNameIn).next := pt;

end if;

Department(deptNameIn).num := Department(deptNameIn).num + 1; -- track number in list

end InsertNode;

empPtr: EmpPt;

begin

insertNodeLeft(Sales, Bob, Sales\_person);

insertNodeLeft(Sales, Mary, Sales\_person);

insertNodeLeft(Sales, Sally, Sales\_person);

insertNodeLeft(Sales, Joe, Sales\_person);

put("list nodes"); new\_line(2);

empPtr := Department(Sales).next;

for index in 1..4 loop

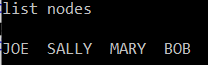
put( empPtr.empName ); put(" ");

empPtr := empPtr.next;

end loop;

null;

end L2Doption;



**Appendix 2:**

**A program showing how to read and write text files follows.**

--RWTextFileEx.adb

with Ada.Text\_IO;

use Ada.Text\_IO;

procedure RWTextFileEx is

package IntIO is new Ada.Text\_IO.Integer\_IO(Integer);

use IntIO;

myArray: array(1..3, 1..3) of integer;

**fileIn, fileOut: File\_Type; --"File\_Type" defined in Text\_IO.**

var:integer;

begin

**Open(File=>fileIn, Mode=> In\_File, Name =>"RawData.txt");**

for row in 1..3 loop

for col in 1..3 loop

**get(fileIn, myArray(row,col) );**

end loop;

end loop;

for row in 1..3 loop

for col in 1..3 loop

put( myArray(row,col), 4 );

end loop;

new\_line;

end loop;

close(File => fileIn);

**Create(File=>fileOut, Mode=> Out\_File, Name => "Result");**

for row in 1..3 loop

for col in 1..3 loop

**put( fileOut, myArray(row,col), 4 );**

end loop;

put(fileOut, ASCII.LF);

end loop;

put(fileOut, "hello world");

Results in file and printed at command line.



**close(fileOut);**

end RWTextFileEx;

RawData.txt

3 4 5

1 2 3

9

7

8

**Appendix 3 – “B” and “A” options:**

-- in file Uabst2C.adb. Demonstrates inheritance to dynamically allocate stacks which can contain

-- heterogeneous values of programmer defined data structures.

--Author: David Burris

--July 15, 2020

--

-- This program utilizes packages AbstStck, MakeCar, and MakePlane.

--

with Ada.Text\_Io; use Ada.Text\_Io;

with Abststck; use Abststck;

with Makecar, Makeplane; use Makecar, Makeplane;

with Ada.Unchecked\_Deallocation;

procedure Uabstst2c is

package IIO is new Ada.Text\_Io.Integer\_Io(Integer); use IIO;

type DepartmentType is (Sales, Crew, Accounting);

-- instantiate Ada.Text\_IO.Enumeration\_IO(DepartmentType); use the I/o package.

type Stack\_Ptr is access Abstractstack; -- pointer to a stack implemented as a linked list.

Newcar, Carpt, Newplane, Planept, Vehiclept: Abstractstackelementptr; -- point to element in stack.

Num: Integer := 2; -- Number of stacks with defalut value - never used.

begin

Put("Enter number of stacks desired: "); Get(Num); -- How many list heads for stacks.

declare

Vehiclestacks: array(1..Num) of Stack\_Ptr; -- Allocatelist array of list head pointers.

-- Potential error. Asssumed sufficient space in stack.

hintStack: array( DepartmentType) of Stack\_Ptr; -- \*\* Hint \*\*

begin

for J in 1..Num loop

Vehiclestacks(J) := new Abstractstack; -- Create create actual list heads

end loop;

Newcar := new Car'(Abstractstackelement with 4, "Ford"); -- Potential error. Asssumed sufficient space in heap.

Push( Vehiclestacks(1), Newcar);

Newplane := new Plane'(Abstractstackelement with 2, 2, "Northrup");

Push(Vehiclestacks(1), Newplane);

Newcar := new Car'(Abstractstackelement with 2, "Chev");

Push(Vehiclestacks(2), Newcar);

for K in 1.. num loop

put("Stack("); put(K,2); put(")"); new\_line;

for I in 1..Stacksize( Vehiclestacks(K).all) loop --(VehicleStack.all) loop

Vehiclept := Pop(Vehiclestacks(K));

if Vehiclept.all in Car then

Identifyvehicle(Car'Class(Vehiclept.all));

elsif Vehiclept.all in Plane then

Identifyvehicle(Plane'Class(Vehiclept.all));

end if;

new\_line;

end loop;

end loop;

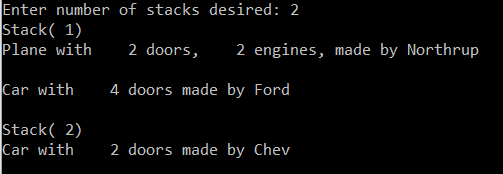
end;

-- All vehicle stack space reclaimed!

-- other work. If loop above "declare, any new stacks could reuse previously used stack space.

-- Heap space for individual stacks should have been reclaimed by programmer prior to leaving

-- the block using "unchecked\_deallocation" if reuse was desired.

****

**Appendix 4:**

**Support files for “A” and “B” options. You are free to utilize code provided in any lab appendix in your lab. This material may be found in “Data Structures Program Class Notes (approximately pages 65 – 71. Generic and inheritance examples appear on pages 54 – 64.**

**Dr. B**

-- Programing by “Classification” (top-down) as opposed to

-- composition by “Composition” (bottom-up).

package AbstStck is

type AbstractStack is limited private;

type AbstractStackElement is tagged private;

type AbstractStackElementPtr is

access all AbstractStackElement'Class;

procedure Push(Stack: access AbstractStack; Y: in AbstractStackElementPtr);

function Pop(Stack: access AbstractStack) return AbstractStackElementPtr;

function StackSize(Stack: AbstractStack) return integer;

private

type AbstractStackElement is tagged

record

Next: AbstractStackElementPtr;

end record;

type AbstractStack is limited

record

Count: integer := 0;

Top: AbstractStackElementPtr := null;

end record;

end AbstStck;

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;

end Pop;

function StackSize(Stack: AbstractStack) return integer is

begin return Stack.Count; end StackSize;

end AbstStck;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 ";

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;

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;

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");

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;

**Sample Output:**

Manufacturer is GMC

Num doors = 2

Manufacturer is Chev

Num doors = 2

Manufacturer is Ford

Num doors = 4

end loop;

end UAbstStc;

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;

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;

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");

push(VehicleStack, NewCar); -- 1st car.

NewPlane := new Plane'(AbstractStackElement with 2, 2, "Northrup");

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!**