CS301/360 PROJECT #2
AUTONOMOUS RESUPPLY VEHICLES

CS301/360, AY 03-2

1. Project Objectives

1.1. **Ada 95 programming constructs.** Demonstrate proficiency with dynamic allocation of memory, linked lists, and array structures.

1.2. **Software development approach.** Apply a logical, structured approach to the development of this program.

2. Project Points

Project # 2 is worth 250 points toward your final course grade.

3. Deadline

Hard and softcopy of your final product must submitted prior to 1500 hours, 6 May 2003. There are two interim submission requirements; see Section 7.

4. Scenario

You are the S-4 for a unit that has recently acquired a rather remarkable technology: autonomous resupply vehicles. These vehicles are capable of navigating their way over any terrain along paths identified by sets of waypoints. As part of your unit's battle plans, you identify a set of 8 depots, one for each resupply vehicle, which serve as each vehicle's start point. They travel their designated paths, distributing supplies to friendly units at each waypoint, and return to their depots for replenishment. (The supplies themselves are secured in containers aboard the vehicles which are locked with cryptokeys, analogous to the ones which secure communications.)

Using manual methods, the task of creating and updating the paths which the vehicles use can be time-consuming, which is problematic during operations when the battlefield requirements are changing rapidly. You have therefore decided to put your education to good use by writing a program that simplifies the management of paths that the vehicles use to accomplish their support mission.

A path consists of a sequence of waypoints, which the vehicles pursue in order, from one waypoint to the next. Vehicles follow a path that is essentially a straight line between points, though they have sensors which allow them to avoid no-go terrain and other obstacles as they find their way from waypoint to waypoint. Each waypoint is characterized by the following attributes:

- A string of at most 10 characters that identifies the name of the waypoint.
- A six-digit grid coordinate. Because of the geographic location of the area of operations, both the northing and easting of any coordinate is in the inclusive range 100 to 999.
- A threat code, which measures the degree of danger associated with passage through that point. The permitted values for threat code are `UNKNOWN`, `GREEN`, `AMBER`, `RED`.

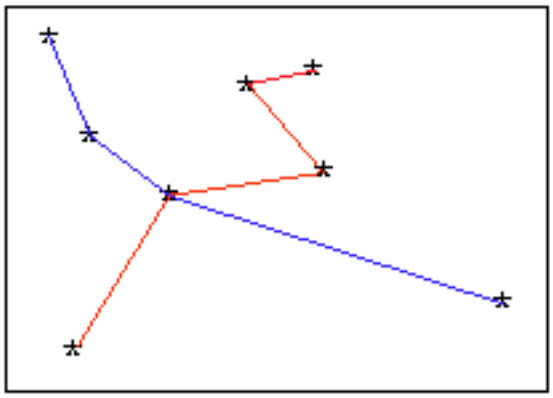
Every non-empty path has a single, well-identified starting waypoint. Each waypoint in a path is connected to at most one subsequent waypoint.

It is possible for more than one path to include a particular waypoint. This has some implications in terms of a requirement to be able to update a waypoint's threat code; if a waypoint's threat code changes for one path, then it should change for all paths. A given waypoint may appear only once in a particular path. No attempt is made to characterize the threat applicable to the traversal between waypoints; only the conditions at each waypoint are of interest.

Paths are stored in a text file called `paths.txt` in the same subdirectory as the main subprogram. Figure 1(a) shows the format of the file. All text in the file may or may not be in upper case, though it appears entirely in upper case here:

```
BEGINPATH
BEGINPOINT
CHARLIE2
144
698
GREEN
ENDPOINT
BEGINPOINT
CHARLIE3
250
821
AMBER
ENDPOINT
   .
   .
   .
ENDPATH
BEGINPATH
   .
   .
   .
ENDPATH
```

(a) Format of the path database text file.



(b) Two paths. Each waypoint is indicated by an asterisk.

FIGURE 1. Representation of paths in the program.

5. REQUIREMENTS

Write a program in Ada95 that would allow you to manage a database of paths in the context identified above. Specifically, your program must afford a user the opportunity to:

1. Create a new path, up to a maximum of 8.
2. Given either a destination coordinate or the name of a waypoint, find the path of *shortest Euclidean distance* that begins at any depot, and which leads to the desired waypoint (if any such path exists). If multiple paths of equal length exist, choose either one arbitrarily. Recall that given the coordinates (x_1, y_1) and (x_2, y_2) we can calculate the straight-line distance between them as $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. In determining the distance of a path, you must follow straight lines connecting each adjacent point on the path; a vehicle may not skip points, even if this would shorten the overall path to the destination waypoint of interest.
3. Given either a destination coordinate or the name of a waypoint, find the *least dangerous* path that begins at any depot, and which leads to the desired waypoint (if any such path exists). In this context a value of `UNKNOWN` should be treated as if it were `RED`, i.e., your assessment is conservative. Likewise, the safety of

an entire path is determined as the single worst threat code encountered at any waypoint on that path. If multiple paths of equal safety exist, choose either one arbitrarily.
4. Add a point to an existing path, at any location within the path.
5. Remove a point from an existing path.
6. Update the threat code of an existing waypoint.
7. Print all paths. You must include each waypoint's ID string, coordinates and threat code.
8. Load a set of paths from the file `paths.txt`.
9. Save the current path database to the file `paths.txt`.

The actions identified here must be implemented in a way that is robust. You must consider the likely ways that a user could mis-enter data, or ways that an unanticipated data interaction could cause your program to fail. Such points of vulnerability should be protected with appropriate exception handlers.

6. Implementation

You must incorporate the following aspects of design into your program's implementation:

1. You must implement the path database as an array of linked lists, one for each of the eight vehicles. Each element of the array is a linked list of elements that refer to waypoints. The order in which elements appear in each list implies the order that the vehicle will travel through the indicated waypoint.
2. You must create a package named `Waypoints` that will include a declaration of the waypoint record type, and all subprograms that include a parameter of that type, or a return value of that type. Certainly other declarations and subprograms may appear in this package as well.

7. Deliverables

7.1. **Interim submission #1 [75 points].** Interim submission #1 must be submitted to the section marcher prior to the start of class on Lesson 29. In it you will express your primary design effort, and a preliminary implementation. It begins with the analysis step of the software development method that we learned at the start of the course, and the first part of the design step: the functional decomposition. For each module that your decomposition reveals, you must supply the following for your interim submission:

1. A plain-text description of *what* the module is intended to do, i.e., what service does it perform.
2. A "black box" diagram showing the inputs to each module as arrows entering a box from the left, and outputs as arrows leaving the box from the right. The box and the arrows are labeled with descriptive names.
3. A plain-text description of each of the inputs and outputs identified in the previous step, and an identification of their data types.
4. A compilable subprogram stub (either a procedure or a function), with parameters corresponding to the previous steps.

In addition to the per-module elements mentioned above, you must supply declarations for any data types and subtypes used as parameters or return types.

7.2. **Interim submission #2 [25 points].** Interim submission #2 must be submitted to the section marcher prior to the start of class on Lesson 33. You must supply the declarations mentioned in Section 7.1, and either code or pseudocode for 3 of the modules that you identified in your functional decomposition. You are free to choose any three modules that you wish.

7.3. **Final submission [150 points].**

7.3.1. *Hard copy (project folder).* A brown project folder is required. Projects must be turned in to the section marcher prior to the start of class on the due date. See the document *CS301 Project Folder Requirements*,which will be provided separately for detailed guidance concerning content and format.

7.3.2. *Soft copy.* Create a directory named `proj2` and ensure all source files for your project are stored there. Submit the contents of your `proj2` directory to the CS301 turn-in directory by doing the following:

1. Change your working directory to the one that *is above* the `proj2` directory. In other words, you should be able to execute an `ls -F` command, and see in the resulting listing an entry `proj2/` (the trailing slash indicates that the entry is a directory).
2. Execute the command `sub cs301 proj2`.