Assignment #4: Darwin

This assignment originated by Nick Parlante and enhanced by Eric Roberts.

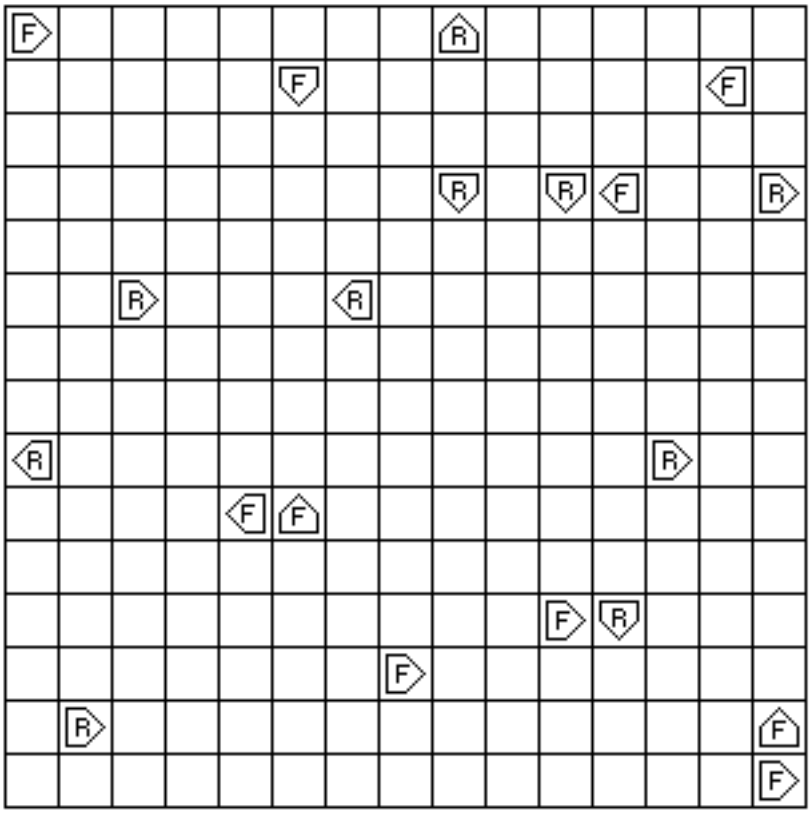
Due: Fri May 5th in class

In this assignment, your job is to build a simulator for a game called *Darwin* invented by Nick Parlante—a game that has become a classic assignment for CS106B. The assignment has a four-fold purpose:

1. To give you a chance to write a large multi-module program.
2. To illustrate the importance of modular decomposition. The entire program is broken down into a series of modules that can be developed and tested independently.
3. To stress the notion of ADTs as a mechanism for sharing data between modules without revealing the representational details.
4. To let you have fun with an application that is extremely captivating and algorithmically interesting in its own right.

The Darwin world

The Darwin program simulates a two-dimensional world divided into squares and populated by a number of *creatures*. Each creature lives in one of the squares, faces in one of the major compass directions (North, East, South, or West) and belongs to a particular *species*, which determines how that creature behaves. For example, one possible configuration of the world is shown below:



The sample world is populated with twenty creatures, ten of a species called *Flytrap* and ten of a species called *Rover*. In each case, the creature is identified in the graphics world with the first letter in its name. The orientation is indicated by the figure surrounding the identifying letter; the

creature points in the direction of the arrow. The behavior of each creature—which you can think of as a small robot—is controlled by a program that is particular to each species. Thus, all of the Rovers behave in the same way, as do all of the Flytraps, but the behavior of each species is different from the other.

As the simulation proceeds, every creature gets a turn. On its turn, a creature executes a short piece of its program in which it may look in front of itself to see what's there and then take some action. The possible actions are moving forward, turning left or right, or *infecting* some other creature standing immediately in front, which transforms that creature into a member of the infecting species. As soon as one of these actions is completed, the turn for that creature ends, and some other creature gets its turn. When every creature has had a turn, the process begins all over again with each creature taking a second turn, and so on. The goal of the game is to infect as many creatures as possible to increase the population of your own species.

Species programming

In order to know what to do on any particular turn, a creature executes some number of instructions in an internal program specific to its species. As an example, consider the Flytrap species program:

step	instruction	comment
1	**`ifenemy 4`**	*If there is an enemy ahead, go to step 4*
2	**`left`**	*Turn left*
3	**`go 1`**	*Go back to step 1*
4	**`infect`**	*Infect the adjacent creature*
5	**`go 1`**	*Go back to step 1*

The step numbers are not part of the actual program, but are included here to make it easier to understand the program. On its turn, a Flytrap first checks to see if it is facing an enemy creature in the adjacent square. If so, the program jumps ahead to step 4 and infects the hapless creature that happened to be there. If not, the program instead goes on to step 2, in which it simply turns left. In either case, the next instruction is a **`go`** instruction that will cause the program to start over again at the beginning of the program.

Programs are executed beginning with the instruction in step 1 and ordinarily continue with each new instruction in sequence, although this order can be changed by certain instructions in the program. Each creature is responsible for remembering the number of the next step to be executed. The instructions that can be part of a Darwin program are listed below:

`hop` The creature moves forward as long as the square it is facing is empty. If moving forward would put the creature outside the boundaries of the world or would cause it to land on top of another creature, the **`hop`** instruction does nothing.

`left` The creature turns left 90 degrees to face in a new direction.

`right` The creature turns right 90 degrees.

`infect` *n* If the square immediately in front of this creature is occupied by a creature of a different species (an "enemy") that creature is infected to become the same as the infecting species. When a creature is infected, it keeps its position and orientation, but changes its internal species indicator and begins executing the same program as the infecting creature, starting at step *n*. The parameter *n* is optional. If it is missing—as it is in the program examples—the new creature should start at the beginning of its new program, so that the **`infect`** instruction with no parameter is equivalent to **`infect 1`**.

ifempty *n* If the square in front of the creature is unoccupied, update the next instruction field in the creature so that the program continues from step *n*. If that square is occupied or outside the world boundary, go on with the next instruction in sequence.

ifwall *n* If the creature is facing the border of the world (which we imagine as consisting of a huge wall) jump to step *n*; otherwise, go on with the next instruction in sequence.

ifsame *n* If the square the creature is facing is occupied by a creature of the same species, jump to step *n*; otherwise, go on with the next instruction.

ifenemy *n* If the square the creature is facing is occupied by a creature of an enemy species, jump to step *n*; otherwise, go on with the next instruction.

ifrandom *n* In order to make it possible to write some creatures capable of exercising what might be called the rudiments of "free will," this instruction jumps to step *n* half the time and continues with the next instruction the other half of the time.

go *n* This instruction always jumps to step *n*, independent of any condition.

A creature can execute any number of **if** or **go** instructions without relinquishing its turn. The turn ends only when the program executes one of the instructions **hop**, **left**, **right**, or **infect**. On subsequent turns, a creature resumes from the point in the program at which it ended its previous turn.

The program for each species is stored in a file in the subfolder named **Creatures** in the assignment folder. Each file in that folder consists of the species name, followed by the steps in the species program, in order. The program ends with the end of file or a blank line. Comments may appear after the blank line or at the end of each instruction line. For example, the program file for the Flytrap creature looks like this:

```
Flytrap
ifenemy 4
left
go 1
infect
go 1

The flytrap sits in one place and spins.
It infects anything which comes in front.
Flytraps do well when they clump.
```

There are several pre-supplied creature files:

Food This creature spins in a square but never infects anything. Its only purpose is to serve as food for other creatures. As Nick Parlante explains, "the life of the **Food** creature is so boring that its only hope in life is to be eaten by something else so that it gets reincarnated as something more interesting."

Hop This creature just keeps hopping forward until it reaches a wall. Not very interesting, but it is useful to see if your program is working.

Flytrap This creature spins in one square, infecting any enemy creature it sees.

Rover This creature walks in straight lines until it is blocked, infecting any enemy creature it sees. If it can't move forward, it turns.

You can also create your own creatures, which will prove helpful for testing.

Overview of the module structure

Your mission in this assignment is to write the Darwin simulator and get it running. It is likely a more challenging task than any assignment you have faced to date. To help manage the complexity, the program has already been broken down into six separate modules as follows:

`darwin`— This module contains the main program, which is responsible for setting up the world, populating it with creatures, and running the main loop of the simulation that gives each creature a turn. The details of these operations are generally handled by the other modules.

`creature`— This module defines an abstract data type representing an individual creature, along with functions for creating new creatures and for taking a turn.

`species`— This module defines an abstract data type representing a species, and provides operations for reading in a species description from a file and for working with the programs that each creature executes.

`world`— This module contains an abstraction for a two-dimensional world, into which you can place the creatures.

`geometry`— This module is already written for you. It defines types to represent points and compass directions, which are the same as those used in the maze program from Chapter 6.

`worldmap`— This module is already written for you. It handles the graphics for the simulation.

As noted above, you are responsible for implementing the first four of the six modules, the remaining two are provided. The structure of each module is described in detail in one of the sections that follow. Even though you have to implement four modules, your task is considerably more manageable because all of the interfaces are specified for you. Moreover, you start with a working implementation in which the missing modules are replaced by **`.lib`** files. This means that you get to play with a working Darwin program immediately beginning on day 1 and can singly swap out our compiled versions and replace them with your implementation to facilitate an incremental development and testing plan.

The geometry module— new types for an x-y-grid

This simple module defines two low-level types (**`pointT`** and **`directionT`**) that are used in several of the other modules. This module contains functions to create new points, compute adjacent points, and rotate directions. The interface file **`geometry.h`** appears in Figure 1 at the end of this handout and provides specifics about the exported types and functions. The code for this module is provided for you and you should not need to make any changes to it.

The worldmap module— graphics for the Darwin world

This module provides the functions necessary to display the creatures on the screen. It exports just two functions, one to draw the initial board, and one to draw the contents of a particular square. The interface file **`worldmap.h`** appears in Figure 2 in the appendix and provides details about using the exported functions. The project folder contains an implementation of this interface. You do not have to write anything for this module, although you should feel free to add embellishments to the graphics code if you have any ideas for interesting extensions.

The darwin module— main program module

The main program module, **`darwin.c`**, is yours to write. There is no interface file for this module since it does not export functions to the other modules. Instead it is the "master control" module which is a client of all the other modules. It is responsible for setting up the world, asking the user which species to set up in the simulation, creating the new creatures at random locations, and running the turn-by-turn simulation until the user clicks to exit.

Hints and requirements for the darwin module:

- The world should have 15 rows by 15 columns. Of course, these should be constants so the size can be easily modified later.
- The user can populate the world with at most 10 species (another opportunity for a constant).
- For each species name the user enters, create 10 of those creatures in the world. If the user wants more of that species, entering the species name again will create another 10, and so on. There can be at most 100 (10 species * 10 each) creatures on the board.
- New creatures should be created in random empty locations, pointing in random directions. To find a location, it's fine to just repeatedly choose a random position and test if it is empty. This will slow down as the board fills up, but this strategy is fine for our purposes.
- Since you deal with user input, your code should be tolerant of the various errors that users make, such as trying to populate the board with an unknown species or too many species.
- Once the simulation starts, your program goes into a loop that allows evolution to progress until the user clicks the mouse to end the simulation.

The world module— abstraction to represent the x-y grid

This module includes the functions necessary to keep track of the creatures in a two-dimensional world. The interface file **`world.h`** appears in Figure 3 of the handout appendix for your perusal. In order for the design to be general, the interface adopts the following design:

1. The world is implemented as an abstract type.
2. The contents are unspecified objects represented as **`void *`** pointers.
3. The dimensions of the world array are specified by the client and therefore must be allocated dynamically.

This general interface allows clients to store any pointer objects—including values of type **`creatureADT`** in Darwin but by no means restricted to that type. This design implies that the internal structure is—at least in part—a two-dimensional dynamic array of **`void *`** pointers.

Hints and requirements for the world module:

- A little hint on creating that array of pointers: a dynamic array with only one dimension is declared as a pointer type, as in

  ```
  int *array;
  ```

  which declares a dynamic array of integers. A two-dimensional dynamic array of integers is implemented as an array of arrays and would be declared like this:

  ```
  int **array;
  ```

  For the worldADT, you need a two-dimensional dynamic array of **`void *`** values rather than integers. The syntax you need for the declaration follows directly from the pattern above, and, yes, there will be three consecutive stars.

- Be careful about handling empty squares. Take care to initialize each element of the **`void *`** matrix to **`NULL`** or some such sentinel and ensure that **`GetContents`** returns **`NULL`** for these unoccupied squares.

- Bulletproof your worldADT against mis-use by having the functions that get and set the location contents range-check their arguments instead just blinding reading out of bounds on the array.

The species module— abstraction to represent each species of creature

The individual creatures in the world are all representatives of some species class and share certain common characteristics, such as the species name and the program they execute. Rather than copy

this information into each creature, this data can be recorded once as part of the description for a species and then each creature can simply include the appropriate species pointer as part of its internal data structure. To encapsulate all of the operations operating on a species within this abstraction, this interface exports a function **`ReadSpecies`** whose job is to read a file containing the name of the creature and its program, as described in the earlier part of this assignment. The remaining functions that allow clients to access information about a species are described in the interface file **`species.h`** in Figure 4 of the handout appendix.

Hints and requirements for the species module:

- The program for a species can contain at most 250 lines.
- To make the folder structure more manageable, the species files for each creature are stored in a subfolder named **`Creatures`**. To open a file in a subfolder, you need to concatenate the string **`":Creatures:"`** (**`"Creatures\\"`** on a PC) onto the beginning of the file name before calling **`fopen`**. You may want to define this prefix as a string constant to make it easy to port your code from one platform to another.
- It is allowable to assume a species program file will be well-formed. However, you might want to aim for a more robust program that halts when it detects a malformed or corrupted instruction.

The creature module— abstraction to represent each individual creature

Creatures are represented as an another abstract type. Each creature is of a particular species and when taking a turn, it is the creature's position in its species program that determines its behavior (i.e. whether to move or turn at this step). All of the actions a creature can take (moving, turning, infecting, etc.) are operations private within the creature module. The exported interface allows the main darwin module to simply ask each creature to "take a turn" and the internals of this module figures out what is the next step for that creature to execute and how to take that action. Please see the **`creature.h`** file in Figure 5 of the handout appendix for the creature interface.

Hints and requirements for the creature module:

- When a new creature is created for a world via the **`NewCreature`** function, this function is responsible for calling the appropriate function in the **`worldmap`** module to add the creature into the specified world.
- Whenever the state of a creature changes—including when it is first initialized—the **`creature`** module is responsible for calling the display function in the **`worldmap`** module to update the graphical representation.

General hints and suggestions

A few suggestions that apply to the program as a whole, rather than any particular module:

- *First, make sure you understand each module and the entire program.* Before you try to implement the modules, it would be worthwhile to study this handout thoroughly and make sure you understand the role of each module and the various functions those modules export.
- *Get each module working before starting on the next one.* Because the interfaces and precompiled library implementations are provided, you should certainly focus on the individual modules rather than the entire program. Do not try to write all the implementations ahead of time and then see if you can get the program working as a whole. Write one module and debug it thoroughly before moving on to the next.
- *Rely on the library modules to simplify debugging.* Suppose that you have completed your implementation of **`darwin.c`** and are ready to move on to **`world.c`**. Instead of leaving your own **`darwin.c`** in the project while you implement the world module, it probably makes more sense to replace it once again with **`darwin.lib`** and then to debug **`world.c`** in that context. As soon as you get it working, you can then run the project with your own versions of both files.

If that doesn't work, one of your modules must be making assumptions about the other modules that the library versions don't.

- *Build your own test cases for creature debugging.* The **`Creatures`** folder supplied with the assignment contains only four creatures: **`Food`**, **`Hop`**, **`Flytrap`**, and **`Rover`**. Although these simple creatures make interesting test cases, they are by no means comprehensive in terms of the features of the Darwin they exercise. To test your program properly, you will need to design new creatures that use a wider range of the Darwin facilities so that you can make sure all the options work. Building the appropriate test cases is an important component of your work as a programmer, and you should not overlook this step in this assignment.

Accessing Files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains these files:

geometry.h	Interface file for geometry module.
geometry.c	Source file which implements the geometry interface.
worldmap.h	Interface file for worldmap module.
worldmap.c	Source file which implements the worldmap interface.
darwin.lib	Compiled library for darwin module.
world.h	Interface file for world module.
world.lib	Compiled library for world module.
species.h	Interface file for species module.
species.lib	Compiled library for species module.
creature.h	Interface file for creature module.
creature.lib	Compiled library for creature module.
Creatures	Folder containing some simple test species.
Darwin Demo	Compiled version of a working program.

To get started, create your own starter project and add the six modules (the .c files for geometry and worldmap and the .lib files for the remaining four modules).At that point, you should have a working solution to play with. Yourwill re-implement the modules **`world`**, **`species`**, **`creature`**, and **`darwin`**, probably (although not necessarily) in that order. As soon as you have written your own implementation of any of these modules, you can remove the **`.lib`** file from the project and add your own **`.c`** file in its place. If you've implemented the module correctly, the program should continue working as a whole.

Deliverables

At the beginning of Friday's lecture, turn a manila envelope containing a printout of your code implementing the four modules and a floppy disk containing your entire project. Everything should be clearly marked with your name, CS106B and your section leader's name! Once again, remember to keep a backup in a safe place. Never turn in the only electronic copy of a program!

Figure 1 geometry.h

```
/* File: geometry.h
 * ----------------
 * This interface provides some extremely simple types and
 * operations useful for manipulating points on an x-y grid.
 */
#ifndef _geometry_h
#define _geometry_h


/* Type: pointT
 * ------------
 * The type pointT is used to encapsulate a coordinate pair into a
 * single value.  Because the record representation makes good
 * intuitive sense and adding an extra level of pointers to the
 * reference would reduce both execution and storage efficiency,
 * this type is exported in its concrete form.
 */
typedef struct {
    int x, y;
} pointT;


/* Type: directionT
 * ----------------
 * This type is an example of an "enumerated type" in C.  The values
 * of type directionT are simply the constants listed in the braces
 * following the enum keyword.  Thus, a variable of type directionT
 * can take on one of the four values North, East, South, and West.
 */
typedef enum { North, East, South, West } directionT;


/* Function: CreatePoint
 * Usage: pt = CreatePoint(x, y);
 * ------------------------------
 * This function combines the x and y coordinates into a pointT
 * structure and returns that value.
 */
pointT CreatePoint(int x, int y);


/* Function: AdjacentPoint
 * Usage: newpt = AdjacentPoint(pt, dir);
 * --------------------------------------
 * This function returns the pointT that results from moving one
 * square in the indicated direction from pt.
 */
pointT AdjacentPoint(pointT pt, directionT dir);


/* Functions: LeftFrom, RightFrom
 * Usage: newdir = LeftFrom(dir);
 * ------------------------------
 * These functions return the directions that result from turning
 * left or right from the given starting direction.
 */
directionT LeftFrom(directionT dir);
directionT RightFrom(directionT dir);
```

Figure 2 worldmap.h

```
/*
 * File: worldmap.h
 * ----------------
 * This interface supports the graphics for the Darwin world.
 */

#ifndef _worldmap_h
#define _worldmap_h

#include "genlib.h"
#include "geometry.h"

/*
 * Function: InitWorldMap
 * Usage: InitWorldMap(columns, rows);
 * -----------------------------------
 * This function opens and displays two windows on the screen, one
 * for the Darwin world and one for the console.  This call must be
 * made before any other calls are made using this package and before
 * any output to the standard I/O channels.  The parameters
 * columns and rows specify the size of the world, although
 * some squares will be outside of the visible display if the
 * world is made too large.
 */

void InitWorldMap(int columns, int rows);

/*
 * Function: DisplaySquare
 * Usage: DisplaySquare(sq, keychar, dir);
 * ---------------------------------------
 * This function changes the display for the indicated square
 * (the location of which is expressed as a point) so that
 * it contains the "creature" indicated by the character keychar
 * facing in the direction specified by dir.  If keychar is a
 * space, the square is displayed as empty and the direction is
 * ignored.
 */

void DisplaySquare(pointT sq, char keychar, directionT dir);

#endif
```

Figure 3 world.h

```
/*
 * File: world.h
 * -------------
 * This interface defines an abstraction which can be used
 * to store objects in an x/y cartesian world.  This abstraction
 * is completely independent of the graphical display, and the
 * client is responsible for any screen updates that are required.
 */

#ifndef _world_h
#define _world_h

#include "genlib.h"
#include "geometry.h"

/*
 * Type: worldADT
 * --------------
 * This abstract type stores the data for a "world," which is
 * defined to be a two-dimensional grid capable of storing
 * arbitrary objects represented as pointers whose type is
 * understood only by the client.
 */

typedef struct worldCDT *worldADT;

/*
 * Function: NewWorld
 * Usage: world = NewWorld(width, height);
 * ---------------------------------------
 * This function creates a new world consisting of width columns
 * and height rows, each of which is numbered beginning at 0.
 * A newly created world contains no objects.
 */

worldADT NewWorld(int width, int height);


/*
 * Function: FreeWorld
 * Usage: FreeWorld(world);
 * ------------------------
 * This function frees all of the storage associated with a world.
 */

void FreeWorld(worldADT world);
```

Figure 3 `world.h` (continued)

```
/*
 * Functions: WorldWidth, WorldHeight
 * Usage: width = WorldWidth(world);
 *        height = WorldHeight(world);
 * ----------------------------------
 * These functions return the width and the height of a world,
 * respectively.
 */

int WorldWidth(worldADT world);
int WorldHeight(worldADT world);

/*
 * Function: InRange
 * Usage: if (InRange(world, pt)) . . .
 * ----------------------------------
 * This function returns TRUE if the specified point pt is within
 * the boundaries  of the world.
 */

bool InRange(worldADT world, pointT pt);

/*
 * Function: SetContents
 * Usage: SetContents(world, pt, obj);
 * ----------------------------------
 * This function places the object obj into the world at the
 * position indicated by pt.
 */

void SetContents(worldADT world, pointT pt, void *obj);

/*
 * Function: GetContents
 * Usage: obj = GetContents(world, pt);
 * ----------------------------------
 * This function returns the object currently in the world at
 * position pt.
 */

void *GetContents(worldADT world, pointT pt);

#endif
```

Figure 4 species.h

```
/*
 * File: species.h
 * ---------------
 * This interface defines the species abstraction.
 */

#ifndef _species_h
#define _species_h

#include "genlib.h"

/*
 * Type: speciesADT
 * ----------------
 * This type is the abstract data type for a species.
 */

typedef struct speciesCDT *speciesADT;

/*
 * Type: opcodeT
 * -------------
 * The type opcodeT is an enumeration of all of the legal
 * command names.
 */

typedef enum {
    Hop, Left, Right, Infect,
    IfEmpty, IfWall, IfSame, IfEnemy, IfRandom,
    Go
} opcodeT;

/*
 * Type: instructionT
 * ------------------
 * The type instructionT is used to represent an instruction
 * and consists of a pair of an operation code and an integer.
 */

typedef struct {
    opcodeT op;
    int address;
} instructionT;
```

Figure 4 `species.h` (continued)

```
/*
 * Function: ReadSpecies
 * Usage: species = ReadSpecies(filename);
 * ---------------------------------------
 * This function reads in a new species from the specified filename.
 * To find the file, the function looks in a subfolder named
 * "Creatures".  If there is no file with the indicated name in
 * that subfolder, the function returns NULL.
 */

speciesADT ReadSpecies(string filename);

/*
 * Function: SpeciesName
 * Usage: name = SpeciesName(species);
 * -----------------------------------
 * This function returns the name for an existing species.
 */

string SpeciesName(speciesADT species);

/*
 * Function: ProgramSize
 * Usage: nSteps = ProgramSize(species);
 * -------------------------------------
 * This function returns the number of instructions in the program
 * for this species.
 */

int ProgramSize(speciesADT species);

/*
 * Function: ProgramStep
 * Usage: statement = ProgramStep(species, k);
 * -------------------------------------------
 * This function returns the kth instruction in the program for
 * this species, where program steps are numbered beginning at 1.
 * Attempting to select an instruction outside the program range
 * generates an error.
 */

instructionT ProgramStep(speciesADT species, int k);

#endif
```

Figure 5 creature.h

```
/*
 * File: creature.h
 * ----------------
 * This interface defines the creature abstraction.
 */

#ifndef _creature_h
#define _creature_h

#include "genlib.h"
#include "geometry.h"
#include "species.h"
#include "world.h"

/*
 * Type: creatureADT
 * -----------------
 * This type is the abstract data type for a creature.
 */

typedef struct creatureCDT *creatureADT;

/*
 * Function: NewCreature
 * Usage: creature = NewCreature(species, world, pt, dir);
 * -------------------------------------------------------
 * This function creates a new creature of the indicated species
 * and places the creature into the specified world.  The creature
 * is initially positioned at position pt facing direction dir.
 */

creatureADT NewCreature(speciesADT species, worldADT world,
                        pointT pt, directionT dir);

/*
 * Function: GetSpecies
 * Usage: species = GetSpecies(creature);
 * --------------------------------------
 * This function returns the species to which this creature
 * belongs.
 */

speciesADT GetSpecies(creatureADT creature);

/*
 * Function: TakeOneTurn
 * Usage: TakeOneTurn(creature);
 * -----------------------------
 * This function executes one turn for this creature.
 */

void TakeOneTurn(creatureADT creature);

#endif
```