3. This question involves reasoning about the code from the Marine Biology Case Study. A copy of the code is provided as part of this exam.

   Consider modifying the Marine Biology Case Study to have fish breed, age, and die. The `Fish` class will have the following changes:

   - A new private data member, `myAge`, will store the age of the fish.
   - A new private data member, `myProbDie`, will store the probability (between 0.0 and 1.0) that the fish dies in any given time step.
   - A new constructor will take the fish's starting age and probability of dying as parameters, in addition to the id and position parameters.
   - The original constructors will set the starting age and probability of dying to default values.
   - A new public member function, `Act`, will take actions for the fish for one step in the simulation.
   - A new private member function, `Breed`, will reproduce new fish.
   - The `Move` function will become a private member function, called by `Act`. (Note that `Simulate::Step` will now call `Fish::Act` rather than `Fish::Move`.)

The modified `Fish` class declaration is shown below with additions in **boldface**.

```
class Fish
{
  public:
    // constructors

    Fish();
    // postcondition: IsUndefined() == true

    Fish(int id, const Position & pos);
    // postcondition: Location() returns pos, Id() returns id,
    //                IsUndefined() == false

    Fish(int id, const Position & pos, int age, double probDie);
    // precondition:  id not used for any other fish;
    //                probDie is between 0.0 and 1.0
    // postcondition: Location() returns pos, Id() returns id,
    //                IsUndefined() == false,
    //                this fish's probability of dying is probDie

    // accessing functions

    int Id() const;
    Position Location() const;
    bool IsUndefined() const;

    apstring ToString() const;
    char ShowMe() const;

    // modifying functions

    void Act(Environment & env);
    // precondition:  this fish is stored in env at Location()
    // postcondition: this fish has moved, bred, or died

  private:

    Neighborhood EmptyNeighbors(const Environment & env,
                                const Position & pos) const;
    void AddIfEmpty(const Environment & env,
                    Neighborhood & nbr, const Position & pos) const;

    void Breed(Environment & env);
    // precondition:  this fish is stored in env at Location();
    //                this fish is old enough to breed
    // postcondition: the neighboring empty positions of this fish have
    //                been filled with new fish, each with age 0 and
    //                the same probability of dying as this fish

    void Move(Environment & env); // now a private member function

    int myId;
    Position myPos;
    bool amIDefined;

    int myAge;          // age of this fish
    double myProbDie;   // probability that this fish dies on a given
                        // step as a probability between 0.0 and 1.0
};
```

The `Environment` class will have the following changes.

- The constructor will read and initialize fish ages and probabilities of dying, along with their positions.
- The `AddFish` member function will take the fish's age and probability of dying as additional parameters.
- A new public member function, `RemoveFish`, will remove an existing fish from the environment.

The modified `Environment` class declaration is shown below with additions in **boldface**.

```
class Environment
{
  public:
    // constructor

    Environment(istream & input);

    // accessing functions

    int NumRows() const;
    int NumCols() const;
    apvector<Fish> AllFish() const;
    bool IsEmpty(const Position & pos) const;

    // modifying functions

    void Update(const Position & oldLoc, Fish & fish);

    void AddFish(const Position & pos, int age, double probDie);
    // precondition:  no fish already at pos, i.e., IsEmpty(pos)
    // postcondition: fish created at pos with the specified age and
    //                probability of dying

    void RemoveFish(const Position & pos);
    // precondition:  there is a fish at pos (IsEmpty(pos) is false)
    // postcondition: fish removed from pos; IsEmpty(pos) is true

  private:
    bool InRange(const Position & pos) const;

    apmatrix<Fish> myWorld;   // grid of fish
    int myFishCreated;        // # fish ever created
    int myFishCount;          // # fish in current environment
};
```

(a) Write the `Environment` member function `RemoveFish`, as started below. `RemoveFish` checks its precondition and prints an error message if the precondition is not met. Otherwise, `RemoveFish` removes the fish in position `pos` from the environment and updates `myFishCount`.

In writing `RemoveFish`, you do not need to include calls to `DebugPrint`.

Complete function `RemoveFish` below.

```
void Environment::RemoveFish(const Position & pos)
// precondition:  there is a fish at pos (IsEmpty(pos) is false)
// postcondition: fish removed from pos; IsEmpty(pos) is true
{

  if (IsEmpty(pos))
  {
    cerr << "error - attempt to remove nonexistent fish at:"
         << pos << endl;
    return;
  }
```

(b) Write the `Fish` member function `Breed`, as started below. `Breed` asks the environment, `env`, to add a new fish in every one of the fish's empty neighboring positions, each with age 0 and with the same probability of dying as this fish.

In writing `Breed`, you do not need to include calls to `DebugPrint`. Assume that all member functions of the `Environment` class work as specified above.

Complete function `Breed` below.

```
void Fish::Breed(Environment & env)
// precondition:  this fish is stored in env at Location();
//                this fish is old enough to breed
// postcondition: the neighboring empty positions of this fish have
//                been filled with new fish, each with age 0 and
//                the same probability of dying as this fish
```

(c) Write the `Fish` member function `Act`, as started below. `Act` will, with probability `myProbDie`, cause the fish to die by calling `env.RemoveFish`. If the fish does not die, it should increment its age. If its new age is three, it should breed; otherwise, it should attempt to move. You will not receive full credit if you reimplement `Move` and `Breed` within function `Act`.

Note: If `r` is defined as follows,

```
RandGen r;
```

then the expression `(r.RandReal() < myProbDie)` will evaluate to `true` with probability `myProbDie`.

In writing `Act`, you do not need to include calls to `DebugPrint`. Assume that all member functions of the `Environment` and `Fish` classes work as specified above. You may also assume that `Environment` member function `RemoveFish` and the `Fish` member function `Breed` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `Act` below.

```
void Fish::Act(Environment & env)
// precondition:  this fish is stored in env at Location()
// postcondition: this fish has moved, bred, or died
```