4. This question involves reasoning about the code from the Marine Biology Case Study. A copy of the code is provided in the Appendix.

   The marine biologists want to study a species of fish that eats algae. Any position in the environment grid can contain zero or more units of algae. If there is any algae at a fish's location, the fish eats one unit of the algae and does not move; otherwise, the fish does not eat. If this is the third consecutive step in which the fish has not eaten, then the fish dies and is removed from the environment. If the fish does not eat and does not die, it moves to a position among its empty neighbors that contains the most algae.

   We represent the algae by adding a matrix of integers to the private data of the `Environment` class. This matrix is the same size as `myWorld`, and each entry represents the number of units of algae at that location. We add three public member functions to the `Environment` class, as well as modifying the `Environment` constructor to initialize `myAlgae`.

```
//  Added to the public section of class Environment

    void RemoveFish(const Position & pos);
    // precondition:  there is a fish at pos
    // postcondition: there is no fish at pos

    int NumAlgaeAt(const Position & pos) const;
    // precondition:  pos is a valid position in the environment
    // postcondition: returns the number of units of algae at pos

    void RemoveAlgae(const Position & pos, int numUnits);
    // precondition:  algae at position pos exceeds numUnits
    // postcondition: algae at position pos has been reduced by numUnits



//  Added to the private section of class Environment

    apmatrix<int> myAlgae;  // number of units of algae at each position
```

GO ON TO THE NEXT PAGE.

We modify the `Fish` class by adding a private data member to keep track of how long since the fish ate any algae. We also add public member function `Act` that encapsulates all the actions of a fish for one step of the simulation and we modify the `Move` function so that the fish moves to the position among its empty neighbors that has the most algae. In order to do this we add private member function `MostAlgae` to the `Fish` class.

```
//  Added to the public section of class Fish

    void Act(Environment & env);
    // precondition:  this Fish is stored in env at Location()
    // postcondition: if there was algae at Location(), this Fish ate
    //                and one unit of algae has been removed from
    //                Location(); otherwise, if this was the third
    //                consecutive step that this Fish did not eat,
    //                then this Fish has been removed from env;
    //                otherwise, this Fish moved.
    //                myStepsSinceFed has been updated.



//  Modified and moved to the private section of class Fish

    void Move(Environment & env);



//  Added to the private section of class Fish

    Position MostAlgae(const Environment & env,
                       const Neighborhood & nbrs) const;
    // precondition:  nbrs.Size() > 0
    // postcondition: returns a Position from nbrs that contains
    //                the most algae.

    int myStepsSinceFed; // steps since this fish last ate
```

(a) Write the `Environment` member function `NumAlgaeAt`, which is described as follows. `NumAlgaeAt` should return the number of units of algae at `pos`.

Complete function `NumAlgaeAt` below.

```
int Environment::NumAlgaeAt(const Position & pos) const
// precondition:  pos is a valid position in the environment
// postcondition: returns the number of units of algae at pos
```

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE A
2003 SCORING GUIDELINES

Question 4

Part A: `NumAlgaeAt` **1 pt**

+1 correct

Part B: `Most Algae` **3 pts**

+1/2 initialize state appropriately (must include index or position, and possibly max, if used)

+1/2 loop over `nbrs`

+1 check for new max in `nbrs`
- **+1/2** attempt (must have `env.NumAlgaeAt(...)`, array syntax does not get attempt) (if `env.` missing, usage note below applies)
- **+1/2** correct

+1/2 update state

+1/2 return position from nbrs determined to contain max algae

Part C: `Act` **5 pts**

+1/2 correct check for algae

Algae present

+1/2 remove algae with call `env.RemoveAlgae(Location(), 1)`

+1/2 reset `myStepsSinceFed`

+1/2 call `env.Update(Location(), *this)` after `myStepsSinceFed` updated

No algae, do fish die?

+1/2 check if this is third step since fed (no steps increment, no point)

+1/2 call `env.RemoveFish(Location())`

Fish doesn't die

+1/2 else (or can use returns on each part) (lose this point if update applies to removed fish)

+1/2 call `Move(env)` (duplicated move logic is OK if correct)

+1/2 increment `myStepsSinceFed`

+1/2 call `env.Update(Location(), *this)` after `myStepsSinceFed` updated

Note: can use `myPos` instead of `Location()`

Usage: -1/2 for once instance of missing `"env."` -1 for two or more missing. (max penalty -1 for problem.)

AP® COMPUTER SCIENCE A
2003 SCORING GUIDELINES

2003 General Usage

Some usage errors may be addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once on a part when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem. If it occurs on different parts of a problem, it should be deducted only once.

Non-penalized errors

case discrepancies, unless confuses identifiers

missing ;'s

missing { }'s where indentation clearly conveys intent

default constructor called with parens, e.g., `BigInt b( )`

`obj.Func` instead of `obj.Func( )`

loop variables used outside loop

`[r, c]` or `(r)(c)` instead of `[r][c]`

= instead of == (and vice-versa)

missing ( )'s around if/while tests

<< instead of >> (and vice-versa)
`*foo.data` instead of `(*foo).data`

Minor errors (1/2 point)

misspelled/confused identifier (e.g., link/next)

no variables declared

void function returns a value

modifying a `const` parameter

unnecessary `cout << "done"`

unnecessary `cin` (to pause)

no * in pointer declaration

`(r,c)` instead of `[r][c]`

use of `L->item` when `L.item` is correct (and conversely)

memory leak due to unneeded node decl (may be taken twice)

Major errors (1 point)

reads new values for parameters (write prompts part of this point)

function result written to output

uses type or class name instead of variable identifier, for example `Fish.move()` instead of `f.move()`

`MemberFunction(obj)` instead of `obj.MemberFunction( )`

`param.FreeFunction( )` instead of `FreeFunction(param)`

Use of object reference that is incorrect or not needed, for example, use of `f.move()` inside member function of `Fish` class

Use of private data when it is not accessible, instead of the appropriate accessor function

destruction of data structure (e.g., by using root ptr for traversal)

Note: Case discrepancies for identifiers fall under the "not penalized" category. However, if they result in another error, they must be penalized. Sometimes students bring this on themselves with their definition of variables. For example, if a student declares "`Fish fish;`", then uses `Fish.move()` instead of `fish.move()`, the one point usage deduction applies.