Advanced Placement Program®

# AP® Computer Science

# Marine Biology Case Study

The College Board
Educational Excellence for All Students

**Advanced Placement Program®**
**Computer Science**

# Marine Biology Case Study

*The AP® Program wishes to acknowledge and to thank Mike Clancy of the University of California, Berkeley, who developed this case study and the accompanying documentation with assistance from Owen Astrachan of Duke University and Cary Matsuoka of Lynbrook High School in San Jose, California.*

# AP Computer Science
# Marine Biology Case Study

## Contents

## Introduction and Background

This case study is a two-part document. Part 1 describes the solution of a programming problem; Part 2 describes the enhancement of a program related to the code designed in Part 1. Each description is written as guidance from an expert to an apprentice about how expertise is applied and decisions are made in the solution. Questions interspersed throughout the text represent places where the expert would ask “What about this alternative?” or say “Now you go try this.” Ideally, you (the reader) will get involved in the solution of each problem and be able to compare your own design and development decisions, along with your understanding of already-written code, with that of the authors.

Included as appendices are listings of the solution code. (This code is also available in the AP section of College Board Online®: www.collegeboard.org/ap/computer-science under “General Course Information”). Each section of the narrative ends with exercises that ask you to analyze or modify existing code, find bugs, apply principles or techniques described in the text to other problems, or reflect on your own programming habits to find ways to improve them.

Both programs treated in the case study are *simulations* — models of real systems. A scientist builds a simulation to better understand a system, to observe and predict its behavior, or to perform experiments on the model. Many real systems are difficult or impossible to observe and control, much less experiment with. A simulation program, however, may be easily run repeatedly and modified to explore the effect of changes to the model.

Many systems involve events that are not precisely predictable. Though the relative frequencies of event results may be known — e.g., a flipped coin comes up heads roughly as often as tails — one can’t be certain of a *particular* result. (Spinners or dice are used in many games to enhance play by incorporating an element of unpredictability.) Such events are simulated in a program using a *random number generator*, a function that on successive calls returns values that appear to be randomly chosen. A class named RandGen, whose member functions produce random values of different types, is provided with the code from these case studies.

The system being simulated is the movement of fish. In Part 1, the movement is back and forth in a narrow aquarium; this is also known as a *one-dimensional random walk*. Part 2 extends the random movement to two dimensions, simulating fish moving in a section of the San Francisco Bay.

# Marine Biology Case Study
# Part 1

## Problem Statement

Biologist friends have approached us to do some software work for them. They wish to install a rectangular fish tank in a nearby aquarium. The tank, which will be built into a wall that separates two rooms, is quite narrow. The biologists would like us to simulate random movement of fish in tanks of different lengths, keeping track of how many times the fish bump into the boundaries of the tank.

The biologists believe that a simple model of fish movement, namely a one-dimensional *random walk*, will yield the information they need. Each fish in the tank will behave as follows:

- it will always be headed either left or right in the tank;
- it will swim one foot in each time unit; and
- it will either continue in the direction it's going or turn completely around, thereby heading toward the opposite end of the tank, with equal probability.

## Preparation

The reader should be familiar with looping and testing constructs in C++, and should have had an introduction to objects and classes. The use of a random number generator is introduced in this part.

## Exercises

**Application** 1. Simulations can be conducted without a computer. For example, one can simulate the movement of a fish by repeatedly flipping a coin. When “heads” is flipped, the fish moves one foot to the right; when “tails” is flipped, it moves one foot to the left. What are the possible positions of the fish, relative to its starting position, at the end of a simulation with six coin flips? Explain. (Don’t worry about the tank being too small.)

**Analysis** 2. Conduct several such simulations, each with six coin flips as described in exercise 1, and keep track of where the fish ends up after each simulation. Is the fish more likely to end up in one position than another? Explain why or why not.

## Simulating a one-dimensional random walk

**How is random behavior simulated in a program?**

The random behavior described by the biologists is a typical component of a simulation program. A program imitates such behavior through the use of a random number generator.* As described in the introduction on page 7, a class named RandGen has been provided for this purpose. Its header file appears in Appendix B.**

**How is the RandGen class used?**

A coin flip has two possible outcomes, heads and tails. It makes sense to represent these as integers, so we focus on the RandInt function. A call to RandInt(2) will work for coin flipping, since it returns either 0 or 1 with equal probability. Following is a client program that simulates six coin flips; we'll call it six-flips.cpp.

```
#include "randgen.h"
#include <iostream.h>

int main()
{
  RandGen randomVals;
  int k;
  int flip;    // the coin value

  for (k = 0; k < 6; k++)
  {
    flip = randomVals.RandInt(2);
    if (flip == 0)
    {
      cout << "heads" << endl;
    }
    else
    {
      cout << "tails" << endl;
    }
  }
  return 0;
}
```

*Stop and help =>*

*Run the program* sixflips.cpp. *Run it again, and compare the output of the second run to that of the first. Does its output look random?*

**How is a seed supplied to the random number generator?**

The RandGen header file mentions the *seed* value used to initialize the random number generation, and points out the advantage of explicitly supplying it when debugging. This is done by supplying the seed value to the RandGen constructor, for example,

```
  RandGen randomVals(100);
```

* More precisely, it's called a *pseudorandom* number generator to reflect the fact that the computer-generated values are not truly random according to mathematical and physical definitions of the term.

** Students are not expected to study or understand (or even look at) the RandGen implementation or its private data members.

Here, the argument 100 (chosen arbitrarily) is used as the seed, thereby ensuring that each run of a program using randomVals produces the same sequence of random values.

*Stop and help =>*

*Modify* sixflips.cpp *by supplying an argument to the* RandGen *constructor. Then run the program a few times to verify that each run produces the same sequence of coin flips.*

**How can a RandGen object be used to implement a random walk?**

The sixflips.cpp program needs only a small amount of modification to implement the "random walk" mentioned in the Introduction. We'll initialize the walker's position to 0; in the loop, each call to RandInt will increase or decrease the position by 1. The program is also made more flexible by having the number of steps depend on user input. We'll refer to the modified client program as onedwalk.cpp; here's its main function.

```
int main()
{
  RandGen randomVals;
  int position = 0;
  int numSteps;
  int step;
  int flip;

  cout << "Number of steps? ";
  cin >> numSteps;
  for (step = 0; step < numSteps; step++)
  {
    flip = randomVals.RandInt(2);
    if (flip == 0)
    {
      position++;
    }
    else
    {
      position--;
    }
  }
  cout << "Final position = "
    << position << endl;
  return 0;
}
```

*Stop and help =>*

*Using a value of 2 for the number of steps, run the* onedwalk.cpp *program several times, keeping track of the final position values. Analyze the results to see if they make sense.*

## Exercises

**Application** 1. Write a client program that uses the RandGen class to simulate the roll of a six-sided die.

**Analysis** 2. A programmer intending to simulate the choice of a color chosen randomly from red, white, and blue writes the following code.

```
apstring color;
if (randomVals.RandInt(3) == 0)
{
  color = "red";
}
else if (randomVals.RandInt(3) == 1)
{
  color = "white";
}
else
{
  color = "blue";
}
```

After numerous test runs, the programmer is surprised to find that the color white is chosen only around 22% of the time, while blue is chosen around 44% of the time. (Red is chosen around 1/3 of the time as expected.) Explain these results.

**Modification** 3. One may also derive a coin flip from a random double. Fill in the blank below with an expression based on the variable r, so that the loop simulates the effect of six coin flips in the same way as the loop in sixflips.cpp.

```
int k;
double r;
int flip;
RandGen randomVals;
for (k = 0; k < 6; k++)
{
  r = randomVals.RandReal();
  flip = ___________ ;
  if (flip == 0)
  {
    cout << "heads" << endl;
  }
  else
  {
    cout << "tails" << endl;
  }
}
```

**Modification** 4. Modify onedwalk.cpp so that it tracks the maximum and minimum positions reached by the walker.

## Simulating fish in a tank

**How can fish behavior be simulated?**

We've simulated a random walk. Now we need to simulate a fish — eventually, more than one fish — doing a random walk. Still to add to the program are the following:

- the tank boundaries: a maximum distance that the fish may go in either direction from the origin;
- counts, for numerous simulated fish, of the number of times they hit the boundaries.

**What design approach is appropriate for the fish simulation?**

This is an excellent situation for applying an object-oriented approach to program design. With this approach, one first organizes the *objects* that will interact in the program. Here, the objects are fish.

Just as a function packages a sequence of actions, an object packages *state* — the object's status, properties, or personal information — with *behavior* — the operations that the object can perform. That's just what's needed here. A fish swims, and is associated with a position and the number of times it bumps against the tank boundaries. A fish object combines the swimming operation and the position and count information in a single place, resulting in a program that's better organized and easier to understand.

**What is a fish's state, and how does it behave?**

We thus proceed to design a C++ class named AquaFish that represents fish objects.

- As just noted, a fish will have a position and a count (the number of times it hit the tank boundaries); we'll name these myPosition and myBumpCount. (It's a good idea to prefix the names of private data members with "my".)
- It will also have a Swim function that changes myPosition and maybe changes myBumpCount.
- It will need a way to report the myBumpCount value at the end of the simulation; we'll have a function named BumpCount to do this.
- Finally, like almost all classes, it will need a constructor that initializes myBumpCount and myPosition.

**How is the Fish class coded?**

Classes in C++ are conventionally coded in two parts. First is the header file that contains the class declaration. It lists the information needed to construct an object, along with the ways the object will interact with other objects. Second is the implementation file, which contains the code for each member function of the class.

Here is a draft version of the AquaFish header file. Following C++ conventions, we place the class declaration in a header file named aquafish.h. The declaration is surrounded by code (ifndef, define, endif) that protects against processing it more than necessary.

```
#ifndef _AQUAFISH_H
#define _AQUAFISH_H

class AquaFish
{
  public:
    AquaFish( ... );       // We don't know the
                           //     arguments yet.
    void Swim();           // Swim one foot.
    int BumpCount() const; // Return the bump
                           //     count.
  private:
    int myPosition;
    int myBumpCount;
};

#endif
```

The file aquafish.cpp is the implementation file. It starts with

```
  #include "aquafish.h"
  #include "randgen.h"
```

and contains definitions for the AquaFish constructor, Swim, and BumpCount. BumpCount is merely an accessing function:

```
  int AquaFish::BumpCount() const
  {
    return myBumpCount;
  }
```

The other functions require more thought.

**How does a fish swim?**

We focus on how a fish swims, figuring that once the Swim function is designed, it will be clear what information must be passed to the constructor.

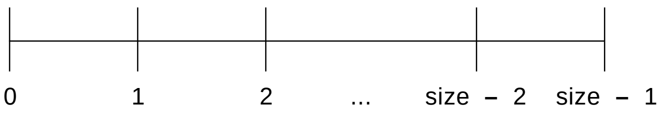
Somewhere in the Swim function will be code similar to the step of a random walk:

```
int flip;
flip = randomVals.RandInt(2);
if (flip == 0)
{
  myPosition++;
}
else
{
  myPosition--;
}
```

Thus, the Swim function needs a RandGen object randomVals. This can either be a private data member or a local variable in Swim; we choose the latter since random number generation is only done in Swim.

**What other information is required to simulate swimming?**

The tank boundaries also must be available to the Swim function. If a fish is at a boundary, it can only move in one direction, so RandInt should not be called. (We assume for now that the size of the tank is an integer number of feet, and will verify this later with the biologists.) A tank could be represented by two integers, one for the leftmost tank position and one for the rightmost position. A simpler representation, however, indicates the leftmost position by 0 and the rightmost position by the size minus 1, as shown in the diagram below.



The tank's left wall is at position 0 and its right wall is at position size – 1. (Note the difference from positions in onedwalk.cpp, where *any* integer was an acceptable position value.) The tank is thus represented by one number rather than two. We'll name the value myTankSize, and use it as follows:

```
if (myPosition == myTankSize - 1)
{
  myPosition--;
}
else if (myPosition == 0)
{
  myPosition++;
}
else
{
  flip = randomVals.RandInt(2);
  // ... (as above)
}
```

If the fish has just bumped into a wall as a result of the move, myBumpCount must be updated:

```
  if (myPosition == 0 ||
      myPosition == myTankSize – 1)
  {
    myBumpCount++;
  }
```

**How is the Fish constructor coded?**

We must have enough information to code the Fish constructor. It must take the tank size as an argument, and will then initialize the position, the bump count, and the private copy of the tank size. We decide for now to start the fish swimming in the middle of the tank, and will check that decision with the biologists later. Here's the code.

```
  AquaFish::AquaFish(int tankSize)
    : myPosition(tankSize/2),
      myTankSize(tankSize),
      myBumpCount(0)
  {
  }
```

**What will be in the main program?**

The main program computes the information useful to the biologists. This information depends on two quantities: the tank size and the number of steps that each simulated fish takes. The program is thus similar to onedwalk.cpp, with three exceptions:

- #include "aquafish.h" replaces #include "randgen.h", and the AquaFish constructor is called instead of the RandGen constructor;
- the program requests and reads the tank size as well as the number of steps; and
- the inside of the loop is replaced by a call to Swim.

We'll put this program in the file aquamain.cpp. Here it is.

```
#include <iostream.h>
#include "aquafish.h"

int main()
{
  int tankSize;
  int stepsPerSim;
  int step;
  cout << "Tank size? ";
  cin >> tankSize;
  cout << "Steps per simulation? ";
  cin >> stepsPerSim;

  AquaFish fish(tankSize);

  for (step = 0; step < stepsPerSim; step++)
  {
    fish.Swim();
  }

  cout << "Bump count = " << fish.BumpCount()
    << endl;

  return 0;
}
```

**What other code must be added to the program?**

One last bit of code is needed. As currently written, the program provides no way to verify that its output is correct. Suppose it says that the bump count is 5; why should this value be believed? There are several possibilities for error.

A list of successive fish positions will provide a way to verify the computed bump count. It will also provide evidence that fish moves are being generated as expected. The biologists probably aren't interested in this information, so the debugging output should be easy to turn off if necessary. We will make it depend on a single boolean variable — we'll call it myDebugging —that's initialized in the AquaFish constructor. Turning the output on or off then requires only a single change.

The complete program appears in Appendix A.

## Exercises

**Analysis** 1. Why isn't

```
#include "randgen.h"
```

needed in the program aquamain.cpp?

**Analysis** 2. Explain how compiling and linking are used with aquafish.cpp, randgen.cpp, and aquamain.cpp to create an executable program.

**Modification** 3. Add an AquaFish constructor that, given a tank size and an initial position for the fish, performs the appropriate initializations.

**Modification** 4. Add an AquaFish constructor that, given a tank size, an initial position, and a boolean value for the debugging switch, performs the appropriate initializations.

**Analysis** 5. The modification of exercise 4 shifts the responsibility for producing debugging output from the AquaFish class to the main program (which provides the debugging value to the AquaFish constructor). Discuss the pros and cons of this shift in responsibility.

**Modification** 6. Modify the AquaFish class to keep track of the maximum and minimum positions that the fish reaches in the tank, and add member functions AquaFish::MaxPos() and AquaFish::MinPos() to report these quantities.

**Debugging, Reflection** 7. Get a friend, a classmate, or a lab partner to introduce a bug into the program by changing one word or number. Then (without comparing the modified version to the original) find the bug. What aspects of this program make debugging difficult?

**Modification** 8. (This exercise presumes familiarity with apvectors.) Using an apvector, modify the AquaFish class so that a fish keeps track of how many times it visits each position. Also add a member function named NumVisits that accesses this vector as described below.

```
int AquaFish::NumVisits(int index)
// precondition:  0 ≤ index < myTankSize.
// postcondition: returns the number of
//                times that the fish
//                visited position index.
```

# Marine Biology Case Study
# Part 2

## Problem Statement

Our biologist friends have approached us to do some more software work for them. They have a diskette containing a program that simulates fish swimming in a small part of the San Francisco Bay. (The program was written several years ago by their research students.) The biologists provide us with the following overview of the program:

- The simulation tracks the movement of fish in a two-dimensional grid.
- In each time step of the simulation, each fish attempts to move to an adjacent grid position.
- The simulation models the prevailing currents in the Bay, leading to certain assumptions about the movement of fish. Unfortunately, the biologists are a bit fuzzy about what these assumptions were.

The diskette contains, in a file named fishsim.cpp, a driver program that runs the simulation. A listing of the program appears in Appendix C.

In earlier research, the biologists studied fish that tended to stay in a limited range of depth and area of the Bay. Now, however, the biologists would also like to investigate the same area with a model that allows them to consider *different* sequences of moving the fish to see if that has a significant effect on the simulation results. They would also like us to consider how to extend the program to study oceanic fish species that range over a much larger region (though still at a limited depth range).

The biologists are not experienced programmers and do not have the time to analyze and modify the program. They want us to do this. The modified program should be able to handle a variety of fish population sizes and should be easy to revise to experiment with different sequences and patterns of fish movement.

## Preparation

The reader should be familiar with random number generation as explained in Part 1 of this case study, as well as classes, objects, files, and one- and two-dimensional arrays (the apvector and apmatrix classes).

## Exercises

**Analysis** 1. In which .h file would you look to find what member functions may be used with the sim object defined in the main function of fishsim.cpp?

**Analysis** 2. Which lines in fishsim.cpp would be changed to use a different data file?

**Modification** 3. Modify fishsim.cpp to run five steps at a time without prompting the user to press return. That is, if the user types 20 in response to the prompt "How many steps?," the modified program should prompt after the 5th, 10th, and 15th steps.

**Reflection** 4. In general, when you are trying to become familiar with a program that you've never seen before, what strategies do you use to learn how the program works?

## Experimenting with the program

**How do programmers try to understand unfamiliar code?**

Programmers are often faced with the task of understanding code written by someone else, so that they may revise it to handle different kinds of data or extend it to produce different behavior. That's our task here. The biologists want the program to be able to model different patterns of fish movement in a variety of environments. We are to familiarize ourselves with the simulation program enough to modify it according to the needs of the biologists.

**What information will be needed in order to make the changes requested?**

This task requires that several kinds of information be gathered.

- How is the program run? What input does it require, and in what format must the input be provided? What output does it produce and how should the output be interpreted?
- How is the environment (i.e., the body of water) represented?
- How are fish represented, and how do they move?

**How should this information be gathered?**

Running the program provides one kind of understanding: what the program does. With enough sample runs, one may be able to infer the connections between input data and program output, and perhaps notice clues about the program's internal processing.

Reading the program provides information about *how* it produces its results — what internal objects are maintained, how they communicate with other objects, and so on. In a large program, however, it is easy to get lost in the details of the code; some sample runs provide a context that will help a programmer understand how the code actually works.

Our first step, then, will be simply to run the program and learn what we can by observing the changing output. We'll then move on to study the documentation about the fish.dat file and make some modifications to the data file.

However, there is one part of the program that we will read right away — the main program. It represents the top level of processing and thus provides "the big picture." It also lays out the processing steps, providing direction for further exploration.

**What information does the fishsim.cpp program provide?**

Examining the fishsim.cpp driver program reveals that it requires a text file (fish.dat) as a data source. Eventually we will need to know the format of this data file to design sample runs. For now, though, we continue reading the main function, noticing the following steps:

a. An Environment object named env is created and initialized from an external data source.

b. A Display object named display is used to "show" the environment. This probably means to display it on the screen.

c. A Simulation object named sim is initialized. Its member function Step is repeatedly given the environment, presumably to execute a step of the simulation, and successive results are displayed.

The diskette provided by the biologists contains a data file with which we can run the program. As expected, the program displays a representation of a grid with simulated fish (represented by letters) moving around.

*Stop and help =>*

*Run the program* fishsim.cpp*, using the file* fish.dat *supplied by your instructor. Form some hypotheses about whether the population changes and how the fish move during a run of the program.*

*Stop and predict =>*

*Design test environments to help confirm or refute the hypotheses you just formed.*

**What questions about fish movement are suggested by the sample run?**

Watching the displayed output, we wonder about several aspects of fish movement.

- Fish seem to move 1 environment cell at each step, as noted by the biologists. Is this always the case? Can a fish ever be blocked from moving?
- Fish seem to wander randomly around the environment. How is a fish move selected when presented with more than one place to move?
- How is a fish's movement affected by the boundary, and by the other fish?
- The environment seems to contain the same fish at the end of the run as at the start. Is this always the case?

We'll keep these questions in mind as we continue exploring the program.

**Where should one look for information about the objects used in** fishsim.cpp**?**

C++ programming conventions suggest that information about how to use Environment, Simulation, and Display objects will be found in the corresponding ".h" files environ.h, simulate.h, and display.h. These files should contain all the information that a client program needs to use the corresponding classes: the public member functions, declarations for their arguments, and documentation about their use. Thus, that's the first place we look.

Neither display.h nor simulate.h provides much information. The environ.h file, however, is much more revealing.

**What information does** environ.h **contain?**

Documentation in environ.h includes descriptions of the various member functions. These will be useful eventually, but we ignore them for now. More important at the moment is the description of the input text file:

```
* The format of the data in the stream
* (probably bound to a text file) is
* as follows:
*   The first line has number of rows, number
*   of columns. Each subsequent line stores
*   row/col positions of a fish.
*   All entries are separated by white space.
* For example,
*
*    rows columns
*    row-pos col-pos
*    row-pos col-pos
*    ..
*    row-pos col-pos
*
* /
```

We examine the fish.dat file and confirm that it seems to follow these specifications.

The documentation also indicates, somewhat cryptically, information about the sequence in which fish are processed:

```
* The Fish in the model/environment are
* accessible via a function that returns a
* vector of all the fish. The Environment
* class guarantees that the fish are stored
* in the vector in top-down, left-to-right
* order. In the grid diagrammed below, fish
* are numbered in the order they will be
* stored in the vector (starting with fish
* at index 0).
*
*  +--------------+
*  | 0          1 |
*  |  2    3      |
*  |            4 |
*  |    5    6    |
*  |7        8    |
*  +--------------+
```

The biologists, we recall, were concerned about the sequence in which fish were processed. This suggests another question to be answered:

- Are the fish indeed processed row by row and left to right within a row, as the comment suggests, or does the simulation process them differently?

*Stop and predict =>*

*Why would the sequence in which fish are processed matter?*

**What experiments should be tried first?**

We start with as simple an experiment as possible that still yields useful results. Having just finished experimenting with a fish moving along a one-dimensional path in Part 1 of this case study, we start here with a 1-by-5 environment (1 row, 5 columns) and a single fish in the middle cell at position (0, 2). (Recall that vector elements are indexed starting at 0.) The results are similar to those of the fish of Part 1: the fish moves one position at each step, its move is apparently chosen at random, and it doesn't move outside the environment. No fish are added during the run.

*Stop and help =>*

*Verify the similarity of fish behavior in the 1-by-5 environment with the behavior of fish in a five-foot tank in the program of Part 1. Why might the fish not behave identically?*

A natural next step is to examine the behavior of a single fish in a two-dimensional environment. We do this. The output produces no surprises. We note the absence of diagonal moves; the fish only moves up, down, left, or right.

*Stop and help =>*

*Run the program several times with a single fish at various positions in a 10-by-10 environment. Keep track of the direction the fish moves at each step, in order to provide evidence about whether each direction is equally likely to be selected. Start the fish in the upper left corner. How does it behave at the environment boundaries?*

**What subtleties arise when processing multiple fish?**

We now try including a second fish in a 1-by-5 environment. The two fish are identified by different letters in the output, making it easy to keep track of them. At first glance, it appears that both fish move each step. The fish never seem to cross paths, or even to occupy the same cell in the environment.

We notice, however, that multiple fish can interfere with one another's movement. For example, consider two fish with an empty cell between them. If two fish are not allowed to occupy the same cell, the fish moving first can move into the empty cell, thereby preventing the other fish from moving there. It's even possible for one fish to keep another from moving at all, as in the configuration below:

```
_ _ A _ B
```

(We use an underscore to represent an empty cell.) If fish A moves to the right before fish B gets a chance to move, B will be blocked.

The question of the sequence in which fish are considered for movement now comes to the fore. The comment in `environ.h` suggests that fish are processed row by row, and from left to right within a row. Some alternatives are the following:

- row by row, from right to left within a row;
- column by column, from top to bottom within a column;
- column by column, from bottom to top within a column;
- random choice of fish in such a way that all fish get processed every step;

Some other sequence is possible as well. Assuming a simple algorithm for fish processing, however, we will narrow our experiments to those just listed.

**How can the fish processing sequence be determined?**

Again we start with simple test values, returning to a 1-by-5 environment. (Since each column contains only one cell, this experiment won't provide any information about move sequences within a column; we can find that out, however, with a 5-by-1 environment.)

**Which choices of environment and fish placement reveal the most information?**

The fish should be placed initially so that it's clear which fish moves first. Fish placed at each end of the environment as shown below don't provide any useful information.

```
A _ _ _ B
```

Fish placed together in the middle may or may not yield useful information. Suppose for example that fish start at

```
_ A B _ _
```

If after one step the fish have both moved left,

```
A B _ _ _
```

fish A must have moved first. Similarly, if both fish move right, fish B must have moved first. Any other possible result is inconclusive, however.

The most information results from starting the fish together at an end of the environment. From an initial configuration of

```
A B _ _ _
```

there are only two possibilities:

```
A _ B _ _     (A moved first)
_ A B _ _     (B moved first)
```

Running this test several times produces consistent results — the leftmost fish (fish A in the examples) always moves first. Trying a similar experiment on a 5-by-1 environment reveals that the uppermost fish moves first.

**What choices of fish placement in a two-dimensional environment reveal the most information?**

On this basis we rule out random choice for fish sequencing. Still to determine is whether fish are processed row by row or column by column.

Again we want as simple a case as possible that yields unambiguous results. A 2-by-2 environment seems a good choice. The fish should restrict each other's movement, so the environment should be almost full of fish. We try an example:

```
A B
C _
```

If processing proceeds row by row, the fish will move in the sequence A, then B, then C. The result after one step will be that A doesn't move (assuming that fish don't move diagonally), B moves down, and C doesn't move, resulting in configuration

```
A _
C B
```

If, on the other hand, fish move column by column, C will move into the only move available for B, resulting in

```
A B
_ C
```

Runs consistently support the row-by-row hypothesis: that fish in the 0th row move first, followed by fish in the 1st row, and so on, and that fish are processed left to right within a row.

## Exercises

**Analysis** 1. Evaluate what evidence is provided by running the program with the following initial environment.

```
A _
B C
```

**Analysis** 2. Starting with the configuration below, predict the results after one step of the simulation. Confirm your answer by running the program using this initial configuration.

```
A B C
D _ E
F G H
```

**Analysis** 3. Run the existing program with a 1000-by-1000 environment that contains a single fish. What happens? Determine the largest environment that the program handles satisfactorily on your computer.

**Analysis** 4. What happens when the program is run with a data file that contains two fish at the same position?

**Analysis** 5. Does the sequence of fish positions in the fish.dat file matter? For example, do the following fish.dat files produce the same program behavior? Briefly explain.

```
2 2     2 2
0 0     1 0
0 1     0 1
1 0     0 0
```

**Application** 6. Write a program to create a fish.dat file. It should first prompt the user for the number of rows and columns in the environment and the number of fish to create. It should then prompt the user for the positions of the fish, and write the data values to the file fish.dat in the format specified in environ.h.

## Learning how the program stores fish

**What areas of flexibility do the biologists want the revised program to provide?**

The biologists would like the program revised so that they can experiment with a number of different features of the simulated environment:

- different fish movement patterns;
- different fish processing sequences;
- fish populations in a variety of sizes, with fish in a variety of initial positions;
- larger environments.

**What did the experiments just conducted show?**

The experiments just conducted show that the existing program already allows fish populations of varying sizes and initial configurations to be input. The Environment constructor reads fish coordinates from a file; if a different fish population is desired, only a corresponding file need be provided. (The biologists may of course need the help of an auxiliary program to create the file.)

The size of the simulated environment — the number of rows and the number of columns — is provided in the same file as the initial positions of the fish themselves. Our experiments involved only relatively small environments. With a larger environment as input, the program may run fine, it may run much more slowly, or it may crash; if it doesn't run satisfactorily, it will need to be fixed. Changing the fish movement pattern — say, to allow diagonal moves — or the fish processing sequence will also require program modifications. Our task now is to find out how to do these things.

**What files comprise the program?**

The program consists of eight code files. We have already examined three of the corresponding header files: simulate.h, display.h, and environ.h. They define the Simulation, Display, and Environment classes respectively. These classes each have only a few member functions, and our experiments either made it clear what the functions did or provided at least enough information for an educated guess. Examination of the remaining files and the comments they contain reveals the information in the table below.

| *file* | *class* | *role* |
|---|---|---|
| fishsim | (none) | main program |
| environ | Environment | models a grid of fish |
| display | Display | displays the environment |
| simulate | Simulation | executes steps of the simulation |
| fish | Fish | represents a fish |
| position | Position | represents a grid position |
| nbrhood | Neighborhood | represents a collection of positions |
| utils | (none) | contains several utility functions |

As one might expect, each file defines at most one class. All the class names are nouns; the naming reflects their roles as types of objects. Our task now is to explore the data and functions packaged in these classes to get enough information to produce a program that satisfies the biologists.

**How should one start trying to understand the program?**

One way to see how all the classes fit together is to trace through the code in the various files to see how the objects interact. We have already determined that an Environment object initializes itself with values from the data file; the Display object displays the Environment; and a Simulation object runs the Environment through the simulation. Understanding the program might begin with analyzing its first step, exploring the details of how the Environment reads data.

For the purposes of fixing the program to satisfy the biologists, however, understanding the details of object initialization will be less important than understanding how the various objects collaborate to execute each step of the simulation. Our experiments have revealed little information about how the Simulation and Environment objects interact. It's also not yet clear how the Fish, Position, and Neighborhood classes fit in. We decide, therefore, to explore the code that manages the fish, namely the Simulation and Environment classes.

**How is the environment implemented?**

We first consult the environ.h file to determine how an environment is implemented. The comment in this file says that an Environment object models a grid of fish. Thus we expect that it will include a *container* object (for example, a vector or a matrix) to hold the collection of fish. Sure enough, the Environment class includes an apmatrix of Fish objects, named myWorld, together with the number of fish contained in the matrix.

```
  apmatrix<Fish> myWorld;
  int myFishCount;
```

**What are alternatives for storing collections of elements?**

Ways to represent any collection of items fall into two categories. One way stores the items explicitly in a vector or some more complicated structure. Representing the collection of integers {3, 1, 4, 9} with this technique, for example, stores the integers as follows:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 9 | | | | | | | |

4 elements

or, to make certain operations more efficient:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 9 | | | | | | | |

4 elements

The other type of representation involves a vector whose *index values* represent the values in the collection. Each possible element of the collection is represented by a different *position* in the vector. Storage of the collection elements is *implicit* rather than explicit.

For example, a set of one-digit integers can be represented with a boolean array indexed from 0 to 9. A value of true in a cell of this array means that the corresponding element is in the set, and a value of false means that the element is not in the set. The set {3, 1, 4, 9} would be represented as follows:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| false | true | false | true | true | false | false | false | false | true |

*Stop and consider =>*

*What operations usually associated with collections of integers would be more efficient with explicitly stored collection values than with a representation of collection values as positions?*

*What operations would be less efficient with the explicitly stored collection values?*

Sometimes a hybrid structure is best for a particular application. For example, instead of true or false in the implicit collection representation, one might store either more information about an element in the collection or some representation of "zero" or "undefined" for elements not in the collection.

**How is the myWorld matrix used?**

We go to the environ.cpp file to find out how the fish container matches up with these two representations. The myWorld matrix is used in every function but InRange. Three of the functions seem especially interesting: AddFish, which adds a fish to the environment; IsEmpty, which checks an environment position; and Update, which updates a fish's location.

AddFish creates a new fish with the given position, using myFishCreated as the next available ID. It puts the new fish at the corresponding position of myWorld:

```
myWorld[pos.Row()][pos.Col()]
  = Fish(myFishCreated, pos);
```

(We know from comments in position.h that the position represents the row and column of a grid position.)

IsEmpty, given a position as argument, checks whether a fish can move there by examining the myWorld entry at that position.

```
if (myWorld[pos.Row()]
      [pos.Col()].IsUndefined())
{
  return true; // pos in grid and no
               // fish at pos
}
return false;
```

IsUndefined is a bit puzzling; we consult the fish.h file for more information. What can an "empty" environment cell contain? Not false or 0 — it must contain a Fish object. Thus the authors of the program created a special undefined object, the equivalent of false in the boolean array described previously.

Update is given an old location oldLoc and a fish that has moved, or changed itself in some other way. It first verifies that oldLoc is in range, then compares the IDs of the fish argument and the fish at oldLoc. (Not surprisingly, fish apparently have unique IDs.) Update then does the following:

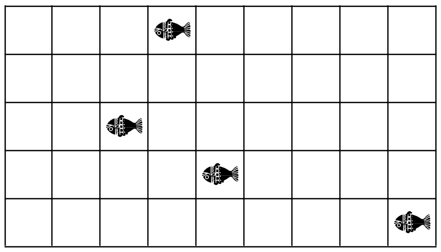
```
// Put an updated copy of fish in fish’s
// current position.
Position newLoc = fish.Location();
myWorld[newLoc.Row()][newLoc.Col()]
  = fish;

// If fish moved, empty out fish’s old
// location.
if (! (oldLoc == newLoc))
{
  myWorld[oldLoc.Row()][oldLoc.Col()]
    = emptyFish;
}
```

Interestingly, each fish is apparently storing its own position. We wonder why this is necessary, and resolve to explore it later. We also note from position.h that the == operator is overloaded but, curiously, the != operator is not. This explains the somewhat clumsy coding of the comparison between oldLoc and newLoc. Finally, we will want to explore how a fish is “moved” prior to the call to Update.

**Which of the two representations for collections does the** myWorld **matrix implement?**

It appears that the myWorld matrix is most similar to the second representation just discussed, an implicit collection of elements represented as index values of nonempty array cells. The program authors took advantage of the fact that two fish are never in the same position simultaneously and used fish position values to index myWorld. The element myWorld[row][col] contains a fish — corresponding to a true value in the above example — if there is a fish at position (row, col). It is undefined — the equivalent of false in the above example — if there is no fish at that position. The diagram below portrays a 5-by-9 myWorld matrix with fish at positions (0, 3), (2, 2), (3, 4), and (4, 8).



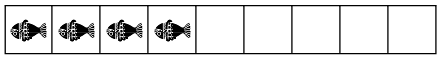
Already we can see one change that will be necessary. A large environment may require a correspondingly large matrix, possibly too large even to be allowed on some computers. A grid with 1000 rows and 1000 columns has a million elements! Some other way must be found to keep track of the fish in such a large environment.

**Where is an alternative representation used?**

Interestingly, the AllFish function uses both collection representations, converting myWorld into a vector named fishList. AllFish contains a loop that copies fish from myWorld into successive elements of fishList:

```
  for (r = 0; r < NumRows(); r++)
  {
    for (c = 0; c < NumCols(); c++)
    {
      if (! myWorld[r][c].IsUndefined())
      {
        fishList[count] = myWorld[r][c];
        count++;
      }
    }
  }
```

This loop is an example of a common programming pattern for processing a matrix row by row, and left to right within a row. The result is shown in the diagram below.



4 fish

Fish, we noted, store their positions; the first fish "knows" it is at position (0, 3), the second at (2, 2), the third at (3, 4), and the fourth at (4, 8), even though their positions in fishList don't indicate this. (Perhaps we've found a reason that a fish stores its position?)

**How is the vector collection of fish used in the rest of the program?**

Curious about how the vector of fish is used, we search the program for places AllFish is called. The Step function in the Simulation class contains a loop that moves the fish:

```
void Simulation::Step(Environment & env)
// postcondition: one step of simulation in
//                env has been made
{
  apvector<Fish> fishList;
  int k;
  fishList = env.AllFish();
  for (k = 0; k < fishList.length(); k++)
  {
    fishList[k].Move(env);
  }
}
```

The Show function in the Display class also calls AllFish. It is not immediately clear how the vector of fish is used there, so we put the Display class aside for a while.

**Which part of the program ensures that the fish are processed in top-to-bottom, left-to-right order?**

AllFish, we realize, is what implements the top-to-bottom, left- to-right sequence of fish activity. The Simulation code calls AllFish to retrieve the fish, then processes them in the same sequence that AllFish provides them. It thus appears that one of the requests of the biologists, the ability to process fish in different sequences, can be satisfied by changing AllFish or the simulation code that calls it.

*Stop and help =>*

*Describe two ways to change the program to have fish processed column by column, left to right, and top to bottom within a column.*

**What has this exploration revealed about the program?**

We can now expand our outline of how the program runs the simulation. Here are the steps.

1. An Environment object initializes itself from a data file; the Display object then displays the environment.
2. A Simulation object repeatedly runs the environment through the simulation. The environment is displayed after each step. In more detail:
   a. The simulation asks the environment for all the fish, ordered top to bottom, left to right.
   b. The simulation asks each fish to move in turn.
   c. The display asks the environment for all the fish; it somehow constructs a corresponding image from them, then displays it.

This is a good start. We have a pretty good idea about how to implement one of the biologists' requests (different fish processing sequences) and we have acquired some information relevant to another (simulation in large environments). Still to be explored are the interaction between the environment and the fish, and the roles of the Position and Neighborhood classes.

## Exercises

**Analysis** 1. Outline the changes that would be necessary to have the top-to-bottom, left-to-right movement sequence be enforced in the Simulation Step function, assuming that that sequence is not the same as the sequence in which the Environment AllFish function returns fish.

**Analysis** 2. List an advantage and a disadvantage of having the Environment AllFish function, rather than the Simulation Step function, control the movement sequence of the fish.

## Learning how fish interact with the environment

**What do** Fish **objects do?**

The interaction between the fish and the environment seems somewhat complicated. We have seen that the Simulation object calls the Move function for each fish, passing it the environment as an argument. Each fish is keeping track of its position in the environment, and the environment is keeping track of fish positions as well. Thus Move must be communicating the fish move to the environment somehow. For a 1-by-N or N-by-1 environment, this interaction is replacing what was a simple call to Swim of Part 1 of this case study. It's time to explore the Fish member functions to find out more about how fish movement is managed.

We start with the fish.h file. A fish is a simple creature; it has an identification number, a position, and an "am I defined" variable. Public member functions include the Move function, two constructors, two functions — Id and Location — that apparently allow access to the data members, an IsUndefined function, and two functions — ShowMe and ToString — that presumably produce displayable representations of a fish.

Moving to the fish.cpp file, we find that most of the functions are short. The IsUndefined function, we recall, was called from the Environment AllFish function to distinguish filled grid positions from empty ones; an "undefined" fish is created by the zero-argument constructor.

## How does a Fish object move?

The most complex function is Move. It's passed an Environment object as its argument. Here is where Neighborhood appears; Move interacts with it as well as with the environment and with Position objects.

The first thing in the Move function is the initialization of a RandGen object as described in Part 1 of this case study. Next, EmptyNeighbors gets called. It considers four neighboring adjacent positions — North, South, East, and West — and adds any of these that are empty to a Neighborhood object, which then is returned to Move. If there are available positions to move to, one is selected at random and becomes the fish's new position. The fish then passes its "moved" self with its new position, along with its old position, to the Environment Update function to record the movement.

It appears that the EmptyNeighbors function implements the fish movement pattern. A change to allow, say, diagonal moves would almost certainly involve this function.

Two questions arise. One: do the various functions called in Move act like we think they do? We consult position.h, nbrhood.h, and environ.h to find out, and are reassured. Two: This seems like a lot of work to get a fish to move. Why is it so difficult?

*Stop and consider =>*

*The task now is to justify the complexity of the interaction of the fish and the environment. Would you continue to explore the code to do this? Why or why not?*

At this point, we decide to step away from the code and think about how we would design the interaction between fish and environment ourselves. This should help us understand the process better, possibly point to ways of simplifying it, and reveal any subtle aspects that we'll need to understand in order to modify the program to handle large environments.

**How might one design the interaction between fish and environment?**

A good way to explore an object-oriented design is to act out the roles of the objects. Pretending to be, say, a fish, can reveal what responsibilities a fish object must have and what it must do to fulfill those responsibilities. We'll do this.

**What does a fish do?**

A fish moves. (Thus something must cause it to move. In the program, the Simulation object does this.) If each fish is supposed to move to a randomly chosen neighboring grid position, this process involves three actions:

1. determining what the neighboring positions are and which of them are candidates for a move;
2. choosing one of the neighboring positions;
3. updating the fish's position.

One might organize the program to have the fish object handle all three steps itself. Generally, however, a good object-oriented design shares responsibility among the various objects rather than having one or two objects that handle everything. Also, one advantage of an object-oriented programming environment is that it allows more accurate simulation of the real world. A real fish, for example, doesn't know its exact coordinates in the body of water in which it's swimming.

**With which objects does the fish communicate?**

In this program, the environment is modeling the collection of fish, and should therefore be able to provide information about the fish's neighbors. It should also be keeping track of which fish is in each position in the grid. For a fish to move, it interacts with the environment in three steps:

1. constructing a neighborhood object that contains possible moves using information from the environment;
2. choosing a random move from the neighborhood;
3. updating the new position of the fish in the environment if it has moved.

This collaboration between fish and environment means that the environment must be accessible to the fish. Either it will be passed as an argument to the Move function or it will be among the fish's private data members.

**How is the fish's position communicated to the environment?**

Role playing, we see how the fish and the environment collaborate to create the neighborhood by constructing a sample dialog.

> Fish (to environment) "Tell me about my neighboring empty squares."
>
> Environment: "Tell me where you are; then I can tell you about your neighbors."

What might the fish answer? In the existing program, the fish essentially says:

> "Here's my position. I have it always by my side."

There are some alternatives, however.

> "I'm Charlie Tuna — surely you know where I am!"
>
> "Here's the position I was just told I was occupying."
>
> "I'm the fish you just told to move — surely you remember me!"

That is, there are several ways for the environment to learn the position of the fish that's moving.

- The fish can store its position.
- The fish can pass the environment some other information that would allow the environment to figure out or look up the fish's position.
- Whoever tells the fish to move can pass the fish its position so that it may in turn pass it to the environment.
- The environment tells the fish to move and keeps track of which fish it has most recently asked to move.

Each of these options has flaws. If the fish and environment are both keeping track of the fish's position, the possibility of inconsistency arises. If the fish asks the environment to determine its position based on other identifying information (such as the fish's ID number), then the environment must do a search through the grid for the fish. If the simulation passes the fish's position to the fish, then the simulation object must somehow know the position (probably by asking the environment to do a search).

In the existing code, the simulation is what tells the fish to move. It doesn't know the fish's position. The program could be modified to have the simulation ask the environment for the position in order to pass it to the fish, or to have it tell the environment what fish is moving. At best, that's more code; at worst, that's a significant restructuring of the program.

It seems that the programmers' decision to have the fish store its own position was a reasonable one. A fish's position is part of its simulated state, even though a fish in real life might not be aware of its position.

**How does a fish choose a new position?**

The dialog just described between the fish and the environment might continue in the following way:

| *Dialog* | *Code* |
|---|---|
| (fish to environment) "Here's my position. Tell me all the directions I can move. Can I move up? What about down? What about left or right?" | `// in Fish::Move`<br>`Neighborhood nbrs = EmptyNeighbors(env, myPos);` |
|  | `// in Fish::EmptyNeighbors`<br>`Neighborhood nbrs;`<br>`AddIfEmpty(env, nbrs, pos.North());`<br>`        ...`<br>`AddIfEmpty(env, nbrs, pos.West());` |
|  | `// in Fish::AddIfEmpty`<br>`if (env.IsEmpty(pos))`<br>`{`<br>`  nbrs.Add(pos);`<br>`}` |
| (fish to itself) "I'll choose [say] the third of these possibilities and make it my new position." | `// in Fish::Move`<br>`RandGen randomVals;`<br>`Position oldPos = myPos;`<br>`myPos = nbrs.Select(randomVals.RandInt(0,`<br>`  nbrs.Size() - 1));` |
| (fish to environment) "I've moved. Update your records to reflect the change."* | `// in Fish::Move`<br>`env.Update(oldPos, *this);  // *this means`<br>`                            //   this fish` |
| (environment to itself) "I need to update my records of where that fish is now and clear out the cell where it used to be." | `// in Environment::Update`<br>`Position newLoc = fish.Location();`<br>`myWorld[newLoc.Row()][newLoc.Col()] = fish;`<br>`if (! (oldLoc == newLoc))`<br>`{`<br>`  myWorld[oldLoc.Row()][oldLoc.Col()]`<br>`    = emptyFish;`<br>`}` |

Interestingly, the environment is relying on each fish to keep track of where it is. At least that eliminates the possibility of a fish thinking it is somewhere and the environment thinking it is somewhere else.

We now believe we understand how a fish moves.

*Stop and help =>* *In your own words, write a summary of the fish movement process.*

* *this is a C++ expression that's not part of the subset in AP CS A; it refers to the fish object itself, and is how information about the fish is transmitted to the environment.

**How is the** Neighborhood **class coded?**

Two components of fish movement remain to be explored. A Neighborhood object, we find, stores positions explicitly in a four-element vector, using one of the representations previously described for a collection of objects. There are no surprises in this code (other than the fact that the size 4 is hard-coded rather than passed as an argument to the constructor). Here's the constructor.

```
Neighborhood::Neighborhood()
  : myList(4),
    myCount(0)
{
}
```

A new position is stored at the end of the neighborhood.

```
if (myCount < myList.length())
{
  myList[myCount] = pos;
  myCount++;
}
```

The Size function keeps track of the number of neighbors. In addition, there is a Select function that retrieves a position given its index, and a ToString function, presumably for debugging, that returns a string representation of the neighborhood contents.

**How is the** Position **class coded?**

The Position class is a straightforward pairing of a row and a column. Movement directions are "hard-coded" into the class via the North, South, East, and West functions. While these functions provide good names for the various movement possibilities — one doesn't have to remember what direction is represented by, say, 2 — they also restrict the movement possibilities. One of the requests of the biologists was the ability to test different fish movement patterns, and the current organization may be difficult to extend to complicated movement rules.

The << and == operators are overloaded to allow easy output and comparison of Position objects. However, the original programmers probably should have overloaded the != operator as well, as we discovered when we were looking at the Environment Update member function. A user of the Position class would expect to be able to use either both operators or neither one.

## Exercises

**Modification** 1. Add the function operator!= to the Position class. Test your modification by using the != operator to compare the two positions in the Environment Update function.

**Analysis** 2. In the design just discussed, the fish chooses a random number and asks for the corresponding neighborhood element. An alternative approach would have the neighborhood include a "return a random neighbor position" operation. What are the advantages and disadvantages of the approach used in the program and this alternative?

**Modification** 3. Make the modification described in the previous exercise and test it thoroughly.

**Modification** 4. Add to the Environment class an EmptyNeighbors function that, given a Position object, returns a Neighborhood object containing adjacent empty positions. Then modify the Fish Move function to call the Environment Empty-Neighbors function instead of querying the environment individually for neighbors. Thoroughly test your revised code.

**Modification** 5. A third alternative is to have the Neighborhood constructor query the Environment about positions adjacent to a given position. Modify the program to implement this alternative. Thoroughly test your revised code.

**Analysis** 6. List advantages and disadvantages of the modifications described in the previous exercises. In particular, who should be in charge of determining possible directions of fish movement, the Fish or the Environment or the Neighborhood?

**Analysis** 7. With a fellow programmer, act out one of the alternative designs for the communication between the fish and the environment. Keep track of all the information that the fish and environment must maintain in the alternative design. Then provide a detailed comparison with the design described in the narrative.

**Reflection** 8. When what seemed to be excessively complex code was encountered in the narrative, the authors resorted to trying to design the code on their own rather than continuing to explore the code. Do you do this? Why or why not? What are the advantages and disadvantages of this tactic for understanding complex code?

## Understanding the rest of the program

**What is the role of the other pieces of the program?**

We have postponed examining the Environment constructor and utils.cpp, and have examined simulate.cpp and display.cpp only in passing. We review these now.

**How does the Environment constructor work?**

The Environment constructor starts by reading the number of rows and columns in the environment, then reads fish positions and calls the AddFish function. It uses a common idiom for handling input, testing the result of the extraction from the input stream:

```
if (input >> numRows >> numCols) ...
while (input >> row >> col) ...
```

**What's in the Simulation class?**

The Simulation class is uncomplicated. Its main function is Step, which we have already examined. It also provides a Run function to allow multiple steps to be run without prompting the user each time.

**How is an environment displayed?**

The Display class, through its Show function, provides a way to display an environment. The version given us displays an environment as text; this facility could have been provided as an Environment member function. The authors probably also anticipated graphical output, however. It might be possible to provide graphical display merely by replacing the Display class, then recompiling the program. Other changes, if necessary, would be made only in the main program, which constructs and uses the Display object.

The Show function starts by getting a vector of fish named fishList from the Environment AllFish function. It then has nested loops that print each cell of the environment, row by row. (This is another example of the row-by-row matrix processing pattern noted in the Environment AllFish function.) An empty cell is printed as a blank; an occupied cell is printed as whatever the Fish ShowMe function returns.

The control is somewhat complicated. Each pair (r, c) of row and column index values is compared with the position of the next fish in fishList. If it matches, ShowMe is called to get a printable version of the fish and the fishIndex variable (used to index fishList) is incremented. This won't work if AllFish were to return fish in a different sequence; an Environment function that, given a position, returns the fish at that position would allow Show to be coded more cleanly.

*Stop and help =>*

*Suppose* AllFish *were to return fish in column-by-column order. Give an example of a fish configuration that would not be properly printed by the* Display Show *function.*

**What's in utils.cpp?**

The utils.cpp file contains three unrelated functions. One, IntToString, returns the apstring representation of a given integer. We recall seeing its use in the ToString functions in the Position and Fish classes.

The second is a Sort function, which we don't remember seeing anywhere before. A search through the rest of the program reveals that it isn't called! We noticed previously that the Environment AllFish function put the fish in the desired processing order. It appears that the program authors also considered an alternative, that of retrieving all the fish and then sorting them by position. When they chose the current implementation, they (accidentally?) left in the Sort function. Maybe we will find it useful when we update the program.

**How does the program produce debugging output?**

Finally comes the DebugPrint function. We had noticed calls to DebugPrint throughout the other classes:

- at the start of the Neighborhood Add function;
- at the start of the Neighborhood Select function;
- in the Fish Move function, after the neighborhood has been created;
- at the end of the Fish Move function;
- at the end of the Environment AllFish function;
- at the end of the Environment InRange function; and
- at the end of the Environment IsEmpty function.

DebugPrint takes an integer argument as well as a string to print. The integer specifies the level of detail at which debugging output is desired. Higher levels mean more detail; the higher the level, the more the output line is indented.

```
void DebugPrint(int level,
                const apstring & msg)
{
  int k;

  if (level <= LEVEL_OF_DEBUG_DETAIL)
  {
    for (k = 0; k < level; k++)
    {
      cout << ' ';
    }
    cout << "**** " << msg << endl;
  }
}
```

We note the following correspondence between messages and detail level:

| *level* | *output* |
|---|---|
| 0 | no debugging output |
| 1 | old and new positions of each moving fish |
| 3 | neighborhood contents |
| 5 | neighborhood element selection index |
| 5 | positions added and not added to the neighborhood |
| 5 | result of AllFish |
| 5 | IsEmpty and InRange calls |

Wow! We are very pleased to see this output facility.

Programmers almost always include debugging output code when first developing a program. All too often, however, they remove it when they think that the program is debugged. This makes it difficult either to track down a bug that was overlooked or to modify the program in any significant way.

Once changes are tested for correctness, the debugging output should be disabled, not eliminated. The levels of debugging output implemented by the authors of the existing code allow the program to be run normally, without debugging output. When debugging output is desired, all that's necessary is to change the debugging level and recompile.

We guess that even-numbered debugging levels were not used to provide more flexibility for maintainers of the program. A modification might require debugging output that's more important than level 5, say, but less important than level 3; thus it would be level 4.

**What events in the program should be displayed?**

In general, it is good to display changes in values of important variables, along with the reasons for those changes. The most important aspect of the program is fish movement, so the authors included output about the old and new positions of each fish at debugging level 1.

Another important variable is the neighborhood. Debugging output should display any position that's added to the neighborhood along with each position selected from the neighborhood. Also, it would be nice to know the reason why a given position is *not* added to the neighborhood. The program authors provided calls to DebugPrint in the Neighborhood functions and also in the Environment InRange and IsEmpty functions. Merely seeing the entire neighborhood from which a selection is made should be enough to narrow down any errors in fish movement, so the debugging output in the Fish Move function has a lower level than output relating to how individual positions get added to the neighborhood.

One final variable of interest is the vector of fish built and returned by AllFish, so it's also displayed.

**How should the debugging output be managed?**

All this output can be difficult to understand unless it's organized. The program authors did this in two ways. One was by assigning different debugging levels to the different events. The other provided a visual organization, by indenting output at a higher level. Visually, the less important output was deemphasized by indenting it. The output appears as follows.

```
important event
  less important event
    minimally important event
  less important event
important event
  less important event
  less important event
    ...
```

One is able to pick out the most important output in the same way as one picks out the main points in an outline.

**Summarize the design.**

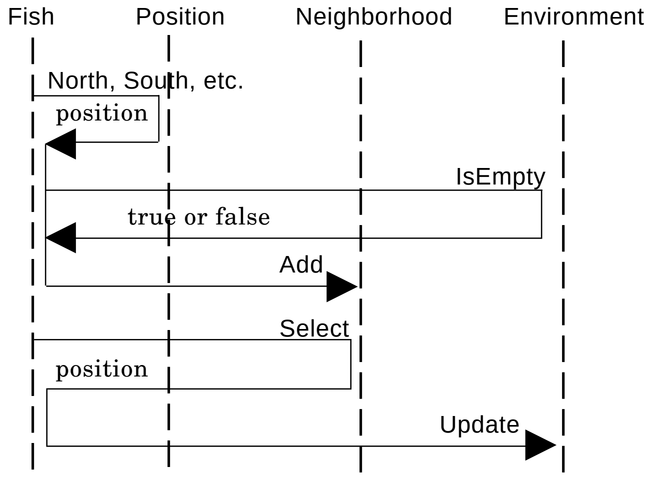
We now have examined all aspects of the program. Here's how its various components fit together.

The main program makes use of Environment, Display, and Simulation objects. It first initializes an environment and displays it, then successively calls the Simulation Step function and displays the resulting environment. Step calls the Environment AllFish function to get a vector of fish; it then tells each fish in the vector to move.

Fish movement involves the Fish, Environment, Neighborhood, and Position classes. It is summarized in the interaction diagram below, which shows how the various classes communicate with one another. (The vertical axis represents time.).

Check each adjacent cell. If it's empty, add it to the neighborhood of empty cells.

Randomly choose one of the empty cells to move to, and inform the environment of the choice.



**How does the design provide or fail to provide the flexibility desired by the biologists?**

The current version of the program already can accommodate fish populations of varying sizes and positions in the environment. More modifications to the program would need to be made to satisfy the biologists' other requests. We have localized the functions to be modified, but will leave the actual modification for a later date.

- The sequence in which fish are processed is determined by the Environment AllFish function, since the Simulation Step function and the Display Show function both process fish in the order AllFish provides them. Changing this sequence requires a change either to AllFish or to the code that calls it.
- The fish movement pattern is determined by the Fish Move function. However, movement in any direction other than horizontally or vertically will probably also require changes to the Position and Neighborhood classes.
- Much of the Environment class is built around a two-dimensional, bounded representation of the environment, and thus will require significant modification to allow experimentation with large environments.

## Exercises

**Modification**

1. Suppose that the Environment class provided a function named FishAt that, given a grid position occupied by a fish, returned the fish at that position. Recode the Display Show function so that, instead of retrieving all the fish at once with AllFish, it queries the environment about each grid position by using IsEmpty and FishAt.

**Analysis, Modification**

2. Suppose the biologists want the order in which fish are moved to be random. Each fish should be moved once and only once per time step, but in a random sequence. What would need to be added or modified in the design to accommodate this change?

**Analysis**

3. Explain how you would produce evidence of the correctness of your modification for exercise 2.

**Analysis, Modification**

4. Suppose that when each fish moved, it was twice as likely to move forward as to any of the sides, and would move backward only when forward or sideways movement was impossible. How should the program be modified to incorporate this new specification?

**Analysis**

5. The Simulation class is quite short. Summarize what role it plays in the program.

**Analysis**

6. Under what conditions would the design of a simple class such as Simulation be preferable to incorporating its operations into other classes?

# APPENDICES

# Appendices Index

# Appendix A

## Program to simulate fish moving left or right in a rectangular tank

### aquafish.h

```
#ifndef _AQUAFISH_H
#define _AQUAFISH_H

class AquaFish
{
  public:
    AquaFish(int tankSize);
    void Swim();                  // Swim one foot.
    int BumpCount() const;        // Return the bump count.
  private:
    int myPosition;
    int myBumpCount;
    int myTankSize;
    bool myDebugging;
};

#endif
```

### aquafish.cpp

```
#include <iostream.h>
#include "aquafish.h"
#include "randgen.h"

AquaFish::AquaFish(int tankSize)
  : myPosition(tankSize/2),
    myTankSize(tankSize),
    myBumpCount(0),
    myDebugging(true)
{

}

void AquaFish::Swim()
{
  RandGen randomVals;
  int flip;

  if (myPosition == myTankSize - 1)
  {
    myPosition--;
  }
  else if (myPosition == 0)
  {
    myPosition++;
  }
  else
  {
    flip = randomVals.RandInt(2);
    if (flip == 0)
    {
      myPosition++;
    }
    else
    {
      myPosition--;
    }
  }

  if (myDebugging)
  {
    cout << "*** Position = " << myPosition << endl;
  }

  if (myPosition == 0 || myPosition == myTankSize - 1)
  {
    myBumpCount++;
  }
}

int AquaFish::BumpCount() const
{
  return myBumpCount;
}
```

## aquamain.cpp

```
#include <iostream.h>
#include "aquafish.h"

int main()
{
  int tankSize;
  int stepsPerSim;
  int step;

  cout << "Tank size? ";
  cin >> tankSize;
  cout << "Steps per simulation? ";
  cin >> stepsPerSim;

  AquaFish fish(tankSize);

  for (step = 0; step < stepsPerSim; step++)
  {
    fish.Swim();
  }

  cout << "Bump count = " << fish.BumpCount() << endl;

  return 0;
}
```

# Appendix B — .h files

## display.h

```
#ifndef _DISPLAY_H
#define _DISPLAY_H

/**
  * Display displays all entries in an environment
  * in a text format, e.g., using fish-supplied characters
  * for fish and space ' ' for empty-space
  *
  * Show(..) is used to display the current state of
  * an environment.  In this text-based display, all information
  * for Show(..) is accessible via the Environment passed into Show(..)
  *
  *
  */

// class declarations for those classes only used
// as references (passed by reference/const reference)

class Environment;

class Display
{
  public:

  // constructor
  Display();
    // postcondition: ready to display an Environment

  // modifying function
  void Show(const Environment & env); // display state of env
    // postcondition: state of env written as text to cout

  private:

    // nothing here now

};

#endif
```

## environ.h

```
#ifndef _ENVIRONMENT_H
#define _ENVIRONMENT_H

/**
  * The Environment class models a grid of fish.
  * An environment is populated at construction time
  * with fish from a stream, presumably bound to a file.
  * The stream is expected to provide data in a certain format
  * as described below.
  *
  * The size of the environment can be determined using the
  * following accessor functions:
  *
```

```
*   - NumRows()
*               the # of rows in the grid
*   - NumCols()
*               the # of columns in the grid
*
* The Fish in the model/environment are accessible via a function
* that returns a vector of all the fish.  The Environment class
* guarantees that the fish are stored in the vector in top-down,
* left-to-right order. In the grid diagrammed below, fish are
* numbered in the order they will be stored in the vector (starting
* with the fish at index 0).
*
*   + — — — — — — — — — — — +
*   | 0                 1  |
*   |  2      3            |
*   |                    4 |
*   |     5    6           |
*   |7            8        |
*   + — — — — — — — — — — — +
*
*   Sample client code for iterating over fish and printing
*   all fish id’s and coordinates in top-down/left-right order
*   appears below.
*
*    // construct environment
*
*    ifstream input("fish.dat");
*    Environment env(input);
*    cout << "grid has dimensions " << env.NumRows() << " x "
*         << env.NumCols() << endl;
*
*    // print fish
*
*    apvector<Fish> fishList = env.AllFish();
*    int k;
*    for (k = 0; k < fishList.length(); k++)
*      cout << fishList[k] << endl;
*
*
* The format of the data in the stream (probably bound to a text file)
* is as follows:
*   The first line has number of rows, number of columns.
*   Each subsequent line stores row/col positions of a fish.
*   All entries are separated by white space.
* For example,
*
*            rows columns
*            row-pos col-pos
*            row-pos col-pos
*            ..
*            row-pos col-pos
*
*/
```

```
#include "fish.h"
#include "apmatrix.h"
#include <fstream.h>

// class declarations for those classes only used
// as references (passed by reference/const reference)

class Position;

class Environment
{
  public:
  // constructor

  Environment(istream & input);
    // precondition:  input is open for reading, in correct format
    // postcondition: environment initialized and populated from input

  // accessing functions

  int NumRows() const;
    // postcondition: returns # rows in grid

  int NumCols() const;
    // postcondition: returns # columns in grid

  apvector<Fish> AllFish() const;
    // postcondition: returned vector (call it fishList) contains all fish
    //                in top-down, left-right order:
    //                top-left fish in fishList[0],
    //                bottom-right fish in fishList[fishList.length()-1];
    //                # fish in environment is fishList.length()

  bool IsEmpty(const Position & pos) const;
    // postcondition: returns true if pos in grid and no fish at pos,
    //                returns false otherwise

  // modifying functions

  void Update(const Position & oldLoc, Fish & fish);
    // precondition:  fish was located at oldLoc, has been updated
    // postcondition: if (fish.Location() != oldLoc) then oldLoc is empty;
    //                Fish fish is updated properly in this environment

  void AddFish(const Position & pos);
    // precondition:  no fish already at pos, i.e., IsEmpty(pos)
    // postcondition: fish created at pos

private:

  bool InRange(const Position & pos) const;
    // postcondition: returns true if pos in grid,
    //                returns false otherwise

  apmatrix<Fish> myWorld;   // grid of fish

  int myFishCreated;        // # fish ever created
  int myFishCount;          // # fish in current environment
};

#endif
```

**fish.h**

```
#ifndef _FISH_H
#define _FISH_H

/**
  * The Fish class represents a fish/swimmer.  A fish has an integer id
  * and maintains its position in a grid. Both these pieces of information
  * are set by the constructor, and the id is never changed.
  * A Fish moves via the member function Move.
  *
  * A fish whose "amIDefined" field is false represents an "empty"
  * or "undefined" fish.  The IsUndefined() member function distinguishes
  * defined fish from undefined fish.  The default Fish() constructor
  * creates an empty/undefined fish.
  */

#include <iostream.h>
#include "apstring.h"
#include "position.h"
#include "nbrhood.h"

// class declarations for those classes only used
// as references (passed by reference/const reference)

class Environment;

class Fish
{
  public:

    // constructors

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

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

    // accessing functions

    int Id() const;
      // precondition:  ! IsUndefined()
      // postcondition: returns id number of fish

    Position Location() const;
      // postcondition: returns current fish position

    bool IsUndefined() const;
      // postcondition: returns true if constructed via default
      //                constructor, false otherwise

    apstring ToString() const;
      // postcondition: returns a stringized form of Fish

    char ShowMe() const;
      // postcondition: returns a character that can make me visible
    // modifying functions

    void Move(Environment & env);
      // precondition: Fish stored in env at Location()
      // postcondition: Fish has moved to a new location in env (if possible)

  private:

    // helper functions

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

    int myId;
    Position myPos;
    bool amIDefined;
};

ostream & operator << (ostream & out, const Fish & fish);
  // postcondition: fish inserted onto stream out

#endif
```

### nbrhood.h

```
#ifndef _NBRHOOD_H
#define _NBRHOOD_H

/**
  * class Neighborhood represents a
  * collection of Positions.
  *
  * Positions can be added to a Neighborhood.
  * Each Position in a Neighborhood is accessible
  * via the functions Select() -- choose a Position --
  * and Size() -- return the # of Positions in a neighborhood.
  *
  * In the current implementation, a maximum of 4 Positions can be
  * added to a neighborhood.  Any call of Add() after the fourth
  * call is ignored.
  *
  */

#include <iostream.h>
#include "position.h"
#include "apvector.h"
#include "apstring.h"

class Neighborhood
{
  public:

    // constructor
    Neighborhood();
      // postcondition: Size() == 0

    // accessing functions

    int Size() const;    // # Positions
      // postcondition: returns # Positions in the neighborhood

    Position Select(int index) const;    // access a Position
      // precondition:  0 <= index < Size()
      // postcondition: returns the index-th Postion in Neighborhood

    apstring ToString() const;    // stringized representation
      // postcondition: returns a string version of all Positions
      //                in Neighborhood

    // modifying functions

    void Add(const Position & pos);      // add pos to neighborhood
      // precondition:  there is room in the neighborhood
      // postcondition: pos added to Neighborhood

  private:

    apvector<Position> myList;
    int myCount;
};

ostream & operator << (ostream & out, const Neighborhood & nbrhood);
  // postcondition: nbrhood inserted onto stream out

#endif
```

## position.h

```
#ifndef _POSITION_H
#define _POSITION_H

/**
  * A Position represents a (row,column) in a grid
  * whose (0,0) is upper-left as in matrix coordinates.
  *
  * Once constructed, a position doesn’t change
  * (all member functions are const) although
  * a Position can be assigned to an existing Position.
  * For example,
  *
  *   Position p(2,3);
  *   Position q;        // default  (-1,-1)
  *   q = p;             // q now at (2,3)
  *
  * Adjacent Positions of a given Position can be
  * determined as illustrated:
  *
  *   Position p(5,5);
  *   Position q;
  *
  *   q = p.North();    // q is (4,5)
  *   q = p.South();    // q is (6,5)
  *   q = p.East();     // q is (5,6)
  *   q = p.West();     // q is (5,4)
  *
  */

#include <iostream.h>
#include "apstring.h"

class Position
{
  public:

    // constructors

    Position();
      // postcondition: Row() == -1, Col() == -1
    Position(int r, int c);
      // postcondition: Row() == r, Col() == c

    // accessing functions

    int Row() const;
      // postcondition: returns row of Position
    int Col() const;
      // postcondition: returns column of Position

    Position North() const;
      // postcondition: returns Position north of (up from) this position
    Position South() const;
      // postcondition: returns Position south of (down from) this position
    Position East() const;
      // postcondition: returns Position east (right) of this position
    Position West() const;
      // postcondition: returns Position west (left) of this position

    bool Equals(const Position & rhs) const;
      // postcondition: returns true iff this position equals rhs

    apstring ToString() const;
      // postcondition: returns stringized form of Position

  private:
    int myRow;
    int myCol;
};

ostream& operator << (ostream & out, const Position & pos);
// postcondition: pos inserted onto stream out

bool operator == (const Position & lhs, const Position & rhs);
// postcondition: returns true iff lhs == rhs

#endif
```

**randgen.h**

```
#ifndef _RANDGEN_H
#define _RANDGEN_H

// RandGen objects provide a source of computer-generated random numbers
// (sometimes known as pseudo-random numbers).
//
// By default, a RandGen object will produce a different series of
// numbers (through repeated calls to the RandInt and RandDouble
// methods) every time the program is run.  When testing, though, it is
// often useful to have a program generate the same sequence of numbers
// each time it is run; this can be achieved by specifying a "seed" when
// the first RandGen object in the program is created.
//
// To construct random integers in a given range, client programs
// should use RandInt.  To construct random doubles, client programs
// should use RandReal.  The ranges for the return values for these
// functions are indicated with mathematical notation:
//    [0..max)     means a number between 0 and max,
//                 not including max but including 0;
//    [0..max]     means a number between 0 and max, including
//                 both 0 and max.
//
// For example,
//    RandGen r;
//    r.RandInt(5)         an int between 0 and 5, not including 5
//    r.RandInt(0, 5)      an int between 0 and 5, might include 0 or 5
//    r.RandReal()         a double between 0 and 1, not including 1
//    r.RandReal(4, 6)     a double between 4 and 6, might include 4 or 6
//
// Technical Note:
//   The "seed" used by all random number generation in a program is set
//   the first time a random number generator object is constructed by
//   the program.  All other random number generator objects created
//   later in the program will use the same seed.

class RandGen
{
  public:

    // Constructors
        // If the first RandGen object is constructed with the default
        // constructor, a different series of numbers is produced every
        // time the program is run.  If the first RandGen object is
        // constructed with a seed, the same series of numbers is produced
        // every time.

    RandGen();                          // default constructor
    RandGen(int seed);                  // produce same series every time
                                        //   (most useful during testing)
    // Accessing functions
        // Ranges for return values are indicated with mathematical
        // notation:
        //    [0..max)     means a number between 0 and max,
        //                 not including max but including 0;
        //    [0..max]     means a number between 0 and max, including
        //                 both 0 and max.

    int RandInt(int max);                     // returns int in [0..max)
    int RandInt(int low, int max);            // returns int in [low..max]
    double RandReal();                        // returns double in [0..1)
    double RandReal(double low, double max); // returns double in
                                              // [low..max]

  private:
    static int ourInitialized;          // for 'per-class' initialization
};

#endif    // _RANDGEN_H not defined
```

**simulate.h**

```
#ifndef _SIMULATION_H
#define _SIMULATION_H

/**
  * Simulation controls a simulation of fish as
  * represented in an Environment.
  *
  * One step of a simulation can be performed via Step(..),
  * an arbitrary number of steps via Run(..).
  *
  */

// class declarations for those classes only used
// as references (passed by reference/const reference)

class Environment;

class Simulation
{
  public:
    Simulation();
    // postcondition: simulation is ready to run

    void Step(Environment & env);
    // postcondition: one step of simulation in env has been made

    void Run(Environment & env, int steps);
    // postcondition: simulation on env run for # steps passed as steps

};

#endif
```

## utils.h

```
#ifndef _UTILS_H
#define _UTILS_H

#include "apstring.h"
#include "apvector.h"
#include "fish.h"

// Collection of useful utility functions

apstring IntToString(int n);
  // postcondition: returns stringized form of n

void Sort(apvector<Fish> & list, int numElts);
  // precondition:  list contains numElts Fish
  // postcondition: list sorted so that entries are
  //                in order top-down/left-right by Position

void DebugPrint(int level, const apstring & msg);
  // print the given debugging error message if level is low enough
  //    (which levels are actually printed can be set in utils.cpp)

#endif
```

# Appendix C — .cpp files

## display.cpp

```
#include <iomanip.h>
#include "display.h"
#include "environ.h"
#include "fish.h"
#include "utils.h"

// constructor

Display::Display()
// postcondition: ready to display an Environment
{
}

// public modifying function

void Display::Show(const Environment & env)
// postcondition: state of env written as text to cout
{
  const int WIDTH = 1; // for each fish

  int rows = env.NumRows();
  int cols = env.NumCols();
  int fishIndex = 0;
  int numFish;
  int r, c;
  Position pos;       // position of fish to be displayed next

  apvector<Fish> fishList;
  fishList = env.AllFish();
  numFish = fishList.length();

  // find position of first fish to be displayed (if any)
  if (fishIndex < numFish)
  {
    pos = fishList[fishIndex].Location();
  }

  for (r = 0; r < rows; r++)
  {
    for (c = 0; c < cols; c++)
    {

      if (pos.Row() == r && pos.Col() == c)
      {
        // this is a position with a fish
        cout << setw(WIDTH) << fishList[fishIndex].ShowMe();
        fishIndex++;
        // find position of next fish to be displayed
        if (fishIndex < numFish)
        {
          pos = fishList[fishIndex].Location();
        }
        else // no more fish to display
        {
          pos = Position(); // not in the grid, won’t be displayed
        }
      }
      else // this position has no fish
      {
        cout << setw(WIDTH) << ' ';
      }
    }  // finished processing all columns in a row
    cout << endl;
  }  // finished processing all rows in the grid
}
```

## environ.cpp

```
#include "environ.h"
#include "fish.h"
#include "utils.h"

// constructor

Environment::Environment(istream & input)
  : myWorld(0,0),
    myFishCreated(0),
    myFishCount(0)
// precondition:  input is open for reading, in correct format
// postcondition: environment initialized and populated from input
{
  int numRows, numCols, row, col;

  if (input >> numRows >> numCols)      // resize the matrix
  {
    myWorld.resize(numRows, numCols);
  }
  else
  {
    cerr << "reading rows/columns failed in Environment" << endl;
    exit(1);
  }
  while (input >> row >> col)
  {
    AddFish(Position(row, col));
  }
}

// public accessing functions

int Environment::NumRows() const
// postcondition: returns # rows in grid
{
  return myWorld.numrows();
}
```

```
int Environment::NumCols() const
// postcondition: returns # columns in grid
{
  return myWorld.numcols();
}

apvector<Fish> Environment::AllFish() const
// postcondition: returned vector (call it fishList) contains all fish
//                 in top-down, left-right order:
//                 top-left fish in fishList[0],
//                 bottom-right fish in fishList[fishList.length()-1];
//                 # fish in environment is fishList.length()
{
  apvector<Fish> fishList(myFishCount);
  int r, c, k;
  int count = 0;
  apstring s = "";

  // look at all grid positions, store fish found in vector fishList

  for (r = 0; r < NumRows(); r++)
  {
    for (c = 0; c < NumCols(); c++)
    {
      if (! myWorld[r][c].IsUndefined())
      {
        fishList[count] = myWorld[r][c];
        count++;
      }
    }
  }

  for (k = 0; k < count; k++)
  {
    s += fishList[k].Location().ToString() + " ";
  }
  DebugPrint(5, "Fish vector = " + s);
  return fishList;
}

bool Environment::IsEmpty(const Position & pos) const
// postcondition: returns true if pos in grid and no fish at pos,
//                 returns false otherwise
{
  if (! InRange(pos))
  {
    return false;  // debug msg printed in InRange
  }

  if (myWorld[pos.Row()][pos.Col()].IsUndefined())
  {
    return true;  // pos in grid and no fish at pos
  }

  DebugPrint(5, pos.ToString() + " contains a fish.");
  return false;
}
```

```
// public modifying functions

void Environment::Update(const Position & oldLoc, Fish & fish)
// precondition:  fish was located at oldLoc, has been updated
// postcondition: if (fish.Location() != oldLoc) then oldLoc is empty;
//                Fish fish is updated properly in this environment
{
  Fish emptyFish;

  if (InRange(oldLoc))
  {
    if (myWorld[oldLoc.Row()][oldLoc.Col()].Id() != fish.Id())
    {
      cerr << "illegal fish move" << endl;
    }
    else
    {
      // Put an updated copy of fish in fish’s current position.
      Position newLoc = fish.Location();
      myWorld[newLoc.Row()][newLoc.Col()] = fish;

      // If fish moved, empty out fish‘s old location.
      if (! (oldLoc == newLoc))
      {
        myWorld[oldLoc.Row()][oldLoc.Col()] = emptyFish;
      }
    }
  }
}

void Environment::AddFish(const Position & pos)
// precondition:  no fish already at pos, i.e., IsEmpty(pos)
// postcondition: fish created at pos
{
  if (! IsEmpty(pos))
  {
    cerr << "error, attempt to create fish at non-empty: " << pos << endl;
    return;
  }
  myFishCreated++;
  myWorld[pos.Row()][pos.Col()] = Fish(myFishCreated, pos);
  myFishCount++;
}

// private helper functions

bool Environment::InRange(const Position & pos) const
// postcondition: returns true if pos is in the grid,
//                returns false otherwise
{
  if (0 <= pos.Row() && pos.Row() < NumRows() &&
      0 <= pos.Col() && pos.Col() < NumCols())
  {
    return true;
  }

  DebugPrint(5, pos.ToString() + " is out of range.");
  return false;
}
```

## fish.cpp

```
#include "fish.h"
#include "environ.h"
#include "randgen.h"
#include "position.h"
#include "nbrhood.h"
#include "utils.h"

// constructors

Fish::Fish()
  : myId(0),
    amIDefined(false)
// postcondition: IsUndefined() == true
{

}

Fish::Fish(int id, const Position & pos)
  : myId(id),
    myPos(pos),
    amIDefined(true)
// postcondition: Location() returns pos, Id() returns id,
//                IsUndefined() == false
{

}

// public accessing functions

int Fish::Id() const
// precondition:  ! IsUndefined()
// postcondition: returns id number of fish
{
  return myId;
}

Position Fish::Location() const
// postcondition: returns current fish position
{
  return myPos;
}

bool Fish::IsUndefined() const
// postcondition: returns true if constructed via default constructor,
//                false otherwise
{
  return ! amIDefined;
}
```

```
apstring Fish::ToString() const
// postcondition: returns a stringized form of Fish
{
  return IntToString(myId) + " " + myPos.ToString();
}

// public modifying functions

void Fish::Move(Environment & env)
// precondition:  Fish stored in env at Location()
// postcondition: Fish has moved to a new location in env (if possible)
{
  RandGen randomVals;
  Neighborhood nbrs = EmptyNeighbors(env, myPos);
  DebugPrint(3, nbrs.ToString());

  if (nbrs.Size() > 0)
  {
    // there were some empty neighbors, so randomly choose one
    Position oldPos = myPos;
    myPos = nbrs.Select(randomVals.RandInt(0, nbrs.Size() - 1));
    DebugPrint(1, "Fish " + ToString() + " moves to " + myPos.ToString());

    env.Update(oldPos, *this);          // *this means this fish

  }
  else
  {
    DebugPrint(1, "Fish " + ToString() + " can’t move.");
  }
}

char Fish::ShowMe() const
// postcondition: returns a character that can make me visible
{
  if (1 <= Id() && Id() <= 26)
  {
    return 'A' + (Id() - 1);
  }
  return '*';
}

// private helper functions

Neighborhood Fish::EmptyNeighbors(const Environment & env,
                                  const Position & pos) const
// precondition:  pos is in the grid being modelled
// postcondition: returns a Neighborhood of pos of empty positions
{
  Neighborhood nbrs;

  AddIfEmpty(env, nbrs, pos.North());
  AddIfEmpty(env, nbrs, pos.South());
  AddIfEmpty(env, nbrs, pos.East());
  AddIfEmpty(env, nbrs, pos.West());
  return nbrs;
}

void Fish::AddIfEmpty(const Environment & env,
                      Neighborhood & nbrs, const Position & pos) const
// postcondition: pos is added to nbrs if pos in env and empty
{
  if (env.IsEmpty(pos))
  {
    nbrs.Add(pos);
  }
}

// free functions

ostream & operator << (ostream & out, const Fish & fish)
// postcondition: fish inserted onto stream out
{
  out << fish.ToString();
  return out;
}
```

## nbrhood.cpp

```
#include "nbrhood.h"
#include "utils.h"

// constructor

Neighborhood::Neighborhood()
  : myList(4),
    myCount(0)
//  postcondition: Size() == 0
{

}

// public accessing functions

int Neighborhood::Size() const
//  postcondition: returns # Positions in the neighborhood
{
  return myCount;
}

Position Neighborhood::Select(int index) const
// precondition:  0 <= index < Size()
// postcondition: returns the index-th Position in Neighborhood
{
  DebugPrint(5, "Selecting neighborhood element # " + IntToString(index));
  return myList[index];
}
```

```
apstring Neighborhood::ToString() const
// postcondition: returns a string version of all Positions in Neighborhood
{
  apstring s = "Neighborhood: ";
  int k;
  for (k = 0; k < myCount; k++)
  {
    s += myList[k].ToString() + " ";
  }
  return s;
}

// public modifying function

void Neighborhood::Add(const Position & pos)
// precondition:  there is room in the neighborhood
// postcondition: pos added to Neighorhood
{
  if (myCount < myList.length())
  {
    DebugPrint(5, "Adding " + pos.ToString() + " to neighborhood.");
    myList[myCount] = pos;
    myCount++;
  }
  else
  {
    DebugPrint(5, "Neighborhood had no room for " + pos.ToString());
  }
}

// free function

ostream & operator << (ostream & out, const Neighborhood & nbrhood)
// postcondition: nbrhood inserted onto stream out
{
  out << nbrhood.ToString();
  return out;
}
```

## position.cpp

```
#include "position.h"
#include "utils.h"

// constructors

Position::Position()
  : myRow(-1),
    myCol(-1)
// postcondition: Row() == -1, Col() == -1
{
}

Position::Position(int r, int c)
  : myRow(r),
    myCol(c)
// postcondition: Row() == r, Col() == c
{
}
```

```
// public accessing functions

int Position::Row() const
// postcondition: returns row of Position
{
  return myRow;
}

int Position::Col() const
// postcondition: returns column of Position
{
  return myCol;
}

Position Position::North() const
// postcondition: returns Position north of (up from) this position
{
  return Position(myRow - 1, myCol);
}

Position Position::South() const
// postcondition: returns Position south of (down from) this position
{
  return Position(myRow + 1, myCol);
}

Position Position::East() const
// postcondition: returns Position east (right) of this position
{
  return Position(myRow, myCol + 1);
}

Position Position::West() const
// postcondition: returns Position west (left) of this position
{
  return Position(myRow, myCol - 1);
}

bool Position::Equals(const Position & rhs) const
// postcondition: returns true iff this position equals rhs
{
  return myRow == rhs.myRow && myCol == rhs.myCol;
}

apstring Position::ToString() const
// postcondition: returns stringized form of Position
{
  apstring s =  "(" + IntToString(myRow) + ","
                + IntToString(myCol) + ")";
  return s;

}
```

```
// free functions

ostream & operator << (ostream & out, const Position & pos)
// postcondition: pos inserted onto stream out
{
  out << pos.ToString();
  return out;
}

bool operator == (const Position & lhs, const Position & rhs)
// postcondition: returns true iff lhs == rhs
{
  return lhs.Equals(rhs);
}
```

### simulate.cpp

```
#include "simulate.h"
#include "apvector.h"
#include "environ.h"

//constructor

Simulation::Simulation()
// postcondition: simulation is ready to run
{

}

// public modifying functions

void Simulation::Step(Environment & env)
// postcondition: one step of simulation in env has been made
{
  apvector<Fish> fishList;
  int k;

  fishList = env.AllFish();
  for (k = 0; k < fishList.length(); k++)
  {
    fishList[k].Move(env);
  }
}

void Simulation::Run(Environment & env, int steps)
// postcondition: simulation on env run for # steps passed as steps
{
  int k;

  for (k = 0; k < steps; k++)
  {
    Step(env);
  }
}
```

## utils.cpp

```
#include "utils.h"
#include "position.h"

apstring IntToString(int n)
// postcondition: returns stringized form of n
{
  if (n == 0)
  {
    return "0";   // special case for 0
  }

  int k;
  apstring reverse = "";    // will be correct, but in reverse
  apstring val = "";        // the string returned

  if (n < 0)                // start with "-" if n < 0
  {
    val = "-";
    n = -n;
  }
  while (n > 0)             // get each digit, catenate in reverse
  {
    reverse += char('0' + n % 10);
    n /= 10;
  }

  // now build the string to return by "unreversing"

  for (k = reverse.length() - 1; k >= 0; k--)
  {
    val += reverse[k];
  }

  return val;
}

void Sort(apvector<Fish> & list, int numElts)
// precondition:  list contains numElts Fish
// postcondition: list sorted so that entries are
//                in order top-down/left-right by Position
{
  // use selection sort

  int j, k, minIndex;
  Position min;
  Position current;
  Fish temp;
  for (j = 0; j < numElts; j++)
  {
    minIndex = j;
    min = list[j].Location();
    for (k = j + 1; k < numElts; k++)
    {
      current = list[k].Location();
      if (current.Row() < min.Row() ||
          (min.Row() == current.Row() && current.Col() < min.Col()))
      {
        min = current;
        minIndex = k;
      }
    }
    temp = list[minIndex];
    list[minIndex] = list[j];
    list[j] = temp;
  }
}

// Indicates level of detail at which we want debugging information.
// 0 => no debugging information displayed
// 1 => fish moves only
// 3 => neighborhood contents + output for 1
// 5 => neighborhood element selection + positions added and not added
//        to the neighborhood + myFish vector + output for 3

const int LEVEL_OF_DEBUG_DETAIL = 0;

// The given msg is to be printed if level (which is positive) is less than
// or equal to LEVEL_OF_BUG_DETAIL, the level of detail at which we want to
// see debugging info.  Indent the printed msg 2 spaces for each level of
// detail.
void DebugPrint(int level, const apstring & msg)
{
  int k;

  if (level <= LEVEL_OF_DEBUG_DETAIL)
  {
    for (k = 0; k < level; k++)
    {
      cout << ' ';
    }
    cout << "**** " << msg << endl;
  }
}
```

## Driver program (fishsim.cpp)

```
#include <iostream.h>
#include <fstream.h>
#include "apstring.h"
#include "environ.h"
#include "display.h"
#include "simulate.h"

int main()
{
  // replace filename below with appropriate file (full path if necessary)
  ifstream input("fish.dat");
  Environment env(input);

  // Display display(100,100);  // for graphics display
  Display display;  // for text display
  apstring s;

  Simulation sim;

  int step;
  int numSteps;

  display.Show(env);

  cout << "--- initialized --- " << endl;

  cout << "How many steps? ";
  cin >> numSteps;
  getline(cin, s);

  for (step = 0; step < numSteps; step++)
  {
    sim.Step(env);
    display.Show(env);
    cout << " step " << step << " (press return)";
    getline(cin, s);
  }

  return 0;
}
```

# Appendix D

## Teacher Support

There are a number of AP publications and videos available to assist AP teachers. For details and ordering information, see the following pages. In addition, many teachers find the following resources to be invaluable:

- **College Board Online® (CBO).** Up-to-date AP information is available via CBO at www.collegeboard.org/ap. From there, you can enter the "Teachers" section, which provides answers to questions about exam dates, costs, and grades; course and exam content; College Board workshops; and much more. You'll also find the latest free-response questions and scoring guidelines, multiple-choice questions, and information about how teachers can join an online discussion group in their subject. The "Technical Corner" is a valuable resource that takes a behind-the-scenes look at who creates the courses and exams; the AP Reading and grading process; the validity and reliability procedures used; and data on student performance. Because of CBO's dynamic nature, and the difficulty of describing it in print, we encourage you to go online and see what's there for yourself.
- **AP workshops and summer institutes.** New and experienced AP teachers are invited to attend workshops and institutes to learn the fundamentals of teaching an AP course, as well as the latest expectations for each course and exam. Sessions ranging from one day to three weeks in length are held year-round. Dates, locations, topics, and fee information are available through the College Board's Regional Offices, in the publication *Graduate Summer Courses and Institutes*, or in the "Teachers" section of our website.
- **AP videoconferences.** Several interactive videoconferences are held each year so that AP teachers can communicate with the high school and college teachers who develop AP courses and exams. Schools that participate in the AP Program are notified of the time, date, and subject of the videoconference in advance. Videotapes of each conference are available shortly after the event.
- **Online discussion groups.** The AP Program has developed an electronic mailing list for each AP subject. Many AP teachers find this free resource to be an invaluable tool for sharing ideas with colleagues on syllabi, course texts, teaching techniques, and for discussing other AP issues and topics. To find out how to subscribe, go to the "Teachers" section of our website.

## Pre-AP™

### Preparing Students for Challenging Courses; Preparing Teachers for Student Success

Pre-AP has two objectives: (1) to promote access to AP for all students; (2) to provide professional development through content-specific strategies to build a rigorous curriculum. Teachers employ Pre-AP strategies and materials to introduce skills, concepts, and assessment methods that prepare students for success when they take AP and other challenging academic courses. Schools use Pre-AP strategies to strengthen and align the curriculum across grade levels, and to increase the academic challenge for all students.

Pre-AP professional development is available to teachers through Building Success workshops and through AP Vertical Teams conferences and workshops.

### Building Success

Building Success is a two-day workshop that assists English and history teachers in designing curricula for grades 7 and above. Teachers learn strategies to help students engage in active questioning, analysis, and constructing arguments. Workshop topics include assessment, interdisciplinary teaching and learning, and vertical planning.

### AP Vertical Teams

These one-day workshops, two-day conferences, and five-day summer institutes enable teams of middle school and high school teachers to prepare Pre-AP students for academic success in AP courses and in college. Topics include organizing effective teams, aligning curricula, and developing content-specific teaching strategies.

For more information about Building Success workshops and for schedules of AP Vertical Teams workshops and conferences, contact your College Board Regional Office. Telephone numbers and addresses appear in the back of this publication. Alternatively, contact Mondy Raibon, Associate Director, AP Program, The College Board, 4330 South MoPac Expressway, Austin, Texas 78735-6734; (512) 891-8404, ext. 127; e-mail: mraibon@collegeboard.org.

## AP Publications and Other Resources

A number of AP publications, CD-ROMs, and videos are available to help students, parents, AP Coordinators, and high school and college faculty learn more about the AP Program and its courses and exams. To sort out those publications that may be of particular use to you, refer to the following key:

| | | | |
|---|---|---|---|
| **Students and Parents** | **SP** | **AP Coordinators and Administrators** | **A** |
| **Teachers** | **T** | **College Faculty** | **C** |

You can order many items online through the College Board Online store at http://cbweb4p.collegeboard.org/tcb/store.html. The most current AP Order Form, which contains information about all available items, can be downloaded from the AP Library (www.collegeboard.org/ap/library). Alternatively, call AP Order Services at (609) 771-7243. American Express, MasterCard, and VISA are accepted for payment.

If you are mailing your order using the AP Order Form, send it to the Advanced Placement Program, Dept. E-05, P.O. Box 6670, Princeton, NJ 08541-6670. Payment must accompany all orders not on an institutional purchase order or credit card, and checks should be made payable to the College Board.

The College Board pays fourth-class book rate postage (or its equivalent) on all prepaid orders; you should allow two to three weeks for delivery. Postage will be charged on all orders requiring billing and/or requesting a faster method of shipment.

Publications may be returned within 15 days of receipt if postage is prepaid and publications are in resalable condition and still in print. Unless otherwise specified, **orders will be filled with the currently available edition**; prices are subject to change without notice.

**AP Bulletin for Students and Parents: Free** **SP**

This bulletin provides a general description of the AP Program, including policies and procedures for preparing to take the exams, and registering for the AP courses. It describes each AP Exam, lists the advantages of taking the exams, describes the grade reporting and award options available to students, and includes the upcoming exam schedule.

**College and University Guide to the AP Program: $10** **C, A**
This guide is intended to help college and university faculty and administrators understand the benefits of having a coherent, equitable AP policy. Topics included are validity of AP grades; developing and maintaining scoring standards; ensuring equivalent achievement; state legislation supporting AP; and quantitative profiles of AP students by each AP subject.

**Course Descriptions: $12** **SP, T, A, C**

Course Descriptions provide an outline of the AP course content, explain the kinds of skills students are expected to demonstrate in the corresponding introductory college-level course, and describe the AP Exam. They also provide sample multiple-choice questions with an answer key, as well as sample free-response questions. A set of Course Descriptions is available for $100. Course Descriptions are also available for downloading free of charge from the AP Library on College Board Online.

**Five-Year Set of Free-Response Questions (1995–99): $5** **T**
(quantities limited)

This is our no-frills publication. Each booklet contains copies of all the free-response questions from the last five exams in its subject; nothing more, nothing less. Collectively, the questions represent a comprehensive sampling of the concepts assessed on the exam in recent years and will give teachers plenty of materials to use for essay-writing or problem-solving practice during the year.

**Interpreting and Using AP Grades: Free** **A, C, T**

A booklet containing information on the development of scoring standards, the AP Reading, grade-setting procedures, and suggestions on how to interpret AP grades.

**Guide to the Advanced Placement Program: Free** **A**

Written for both administrators and AP Coordinators, this guide is divided into two sections. The first section provides general information about the AP Program, such as how to organize an AP Program, the kind of training and support that is available for AP teachers, and a look at the AP Exams and grades. The second section contains more specific details about testing procedures and policies and is intended for AP Coordinators.

**Released Exams: $20**
**($30 for "double" subjects: Calculus, Computer Science, Latin, Physics)** **T**

About every four years, on a staggered schedule, the AP Program releases a complete copy (multiple-choice and free-response sections) of each exam. In addition to providing the multiple-choice questions and answers, the publication describes the process of scoring the free-response questions and includes examples of students' actual responses, the scoring standards, and commentary that explains why the responses received the scores they did.

*Packets of 10: $30.* For each subject with a released exam, you can purchase a set of that year's exams for use in your classroom (e.g., to simulate an AP Exam administration).

**Secondary School Guide to the AP Program: $10** **A, T**

This guide is a comprehensive consideration of the AP Program. It covers topics such as: developing or expanding an AP program; gaining faculty, administration, and community support; AP grade reports, their use and interpretation; AP Scholar Awards; receiving college credit for AP; AP teacher training resources; descriptions of successful AP programs in nine schools around the country; and "Voices of Experience," a collection of ideas and tips from AP teachers and administrators.

**Student Guides:**
**(available for Calculus, English, and U.S. History): $12** **SP**

These are course and exam preparation manuals designed for high school students who are thinking about or taking a specific AP course. Each guide answers questions about the AP course and exam, suggests helpful study resources and test-taking strategies, provides sample test questions with answers, and discusses how the free-response questions are scored.

**Teacher's Guides: $12** **T**

Whether you're about to teach an AP course for the first time, or you've done it for years but would like to get some fresh ideas for your classroom, the Teacher's Guide can be your adviser. It contains syllabi developed by high school teachers currently teaching the AP course and college faculty who teach the equivalent course at their institution. Along with detailed course outlines and innovative teaching tips, you'll also find extensive lists of recommended teaching resources.

**AP Vertical Team Guides** **T, A**

An AP Vertical Team (APVT) is made up of teachers from different grade levels who work together to develop and implement a sequential curriculum in a given discipline. The team's goal is to help students acquire the skills necessary for success in AP. To help teachers and administrators who are interested in establishing an APVT at their school, the College Board has published three guides: *AP Vertical Teams in Science, Social Studies, Foreign Language, Studio Art, and Music Theory: An Introduction* ($12); *A Guide for Advanced Placement English Vertical Teams* ($10); and *Advanced Placement Program Mathematics Vertical Teams Toolkit* ($35). A discussion of the English Vertical Teams guide, and the APVT concept, is also available on a 15-minute VHS videotape ($10).

**EssayPrep™** **SP, T**

EssayPrep is available through the AP subject pages of College Board Online. Students can select an essay topic, type a response, and get an evaluation from an experienced reader. The service is offered for the free-response portions of the AP Biology, English Language and Composition, English Literature and Composition, and U.S. History exams. The fee is $15 per response for each evaluation. SAT II: Writing topics are also offered for a fee of $10. Multiple evaluations can be purchased at a 10-20% discount.

**The College Handbook with**
**College Explorer® CD-ROM: $25.95** **SP, T, A, C**

Includes brief outlines of AP placement and credit policies at two- and four-year colleges across the country. Also noted is the number of freshmen granted placement and/or credit for AP in the prior year.

**APCDs™: $49 (home version),**
**$450 (multi-network site license)** **SP, T**

These CD-ROMs are currently available for Calculus AB, English Language, English Literature, European History, Spanish Language, and U.S. History. They each include several actual AP Exams, interactive tutorials, and other features including exam descriptions, answers to frequently asked questions, study skill suggestions, and test-taking strategies. There is also a listing of resources for further study and a planner for students to schedule and organize their study time.

**Videoconference Tapes: $15** **SP, T, C**

Each year, AP conducts interactive videoconferences for various subjects, enabling AP teachers and students to talk directly with the Development Committees that design the AP Exams. Tapes of these events are available in VHS format and are approximately 90 minutes long.

**AP Pathway to Success**
**(video — available in English and Spanish): $15** **SP, T, A, C**

This 25-minute video takes a look at the AP Program through the eyes of people who know AP: students, parents, teachers, and college admissions staff. They answer such questions as “Why Do It?” “Who Teaches AP Courses?” and “Is AP For You?” College students discuss the advantages they gained through taking AP, such as academic self-confidence, writing skills, and course credit. AP teachers explain what the challenge of teaching AP courses means to them and their school, and admissions staff explain how they view students who have stretched themselves by taking AP Exams. There is also a discussion of the impact that an AP program has on an entire school and its community, and a look at resources available to help AP teachers, such as regional workshops, teacher conferences, and summer institutes.

**What’s in a Grade? (video): $15** **T, C**

AP Exams are composed of multiple-choice questions (scored by computer), and free-response questions that are scored by qualified professors and teachers. This video presents a behind-the-scenes look at the scoring process, featuring footage shot on location at the 1992 AP Reading at Clemson University and other Reading sites. Using the AP European History Exam as a basis, the video documents the scoring process. It shows AP faculty consultants in action as they engage in scholarly debate to define precise scoring standards, then train others to recognize and apply those standards. Footage of other subjects, interviews with AP faculty consultants, and explanatory graphics round out the video.

## College Board Regional Offices

*National Office*
Lee Jones/Phil Arbolino/Robert DiYanni/Michael Johanek/Frederick Wright/Trevor Packer
45 Columbus Avenue
New York, NY 10023-6992
E-mail: ap@collegeboard.org
(212) 713-8066

*Middle States*
Mary Alice McCullough/Michael Marsh
3440 Market Street, Suite 410
Philadelphia, PA 19104-3338
(215) 387-7600

*Midwestern*
Bob McDonough/Paula Herron/Ann Winship
1560 Sherman Avenue, Suite 1001
Evanston, IL 60201-4805
(847) 866-1700

*New England*
Fred Wetzel
470 Totten Pond Road
Waltham, MA 02451-1982
(781) 890-9150

*Southern*
Tom New
100 Crescent Centre Parkway, Suite 340
Tucker, GA 30084-7039
(770) 908-9737

*Southwestern*
Frances Brown/Mondy Raibon/Scott Kampmeier/Paul Sanders
4330 South MoPac Expressway, Suite 200
Austin, TX 78735-6734
(512) 891-8400

*Dallas/Fort Worth Metroplex AP Office*
Kay Wilson
P.O. Box 19666, 600 South West Street, Room 108
Arlington, TX 76019
(817) 272-7200

*Western*
Claire Pelton/Gail Chapman
2099 Gateway Place, Suite 480
San Jose, CA 95110-1048
(408) 452-1400

*Canada (AP Program Only)*
George Ewonus
Suite 212-1755 Springfield Road
Kelowna, B.C., Canada V1Y 5V5
(250) 861-9050
E-mail: gewonus@ap.ca

Staff at U.S. College Board Regional Offices other than the National Office can be reached via e-mail using their first initial and last name@collegeboard.org

I.N. 987510