AP Computer Science A
2001 Free-Response Questions

COMPUTER SCIENCE A
SECTION II

Time—1 hour and 45 minutes

Number of questions—4

Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN C++.

Note: Assume that the standard libraries (e.g., `iostream.h`, `fstream.h`, `math.h`, etc.) and the AP C++ classes are included in any program that uses a program segment you write. If other classes are to be included, that information will be specified in individual questions. Unless otherwise noted, assume that all functions are called only when their preconditions are satisfied. A Quick Reference to the AP C++ classes is included in the case study insert.

GO ON TO THE NEXT PAGE.

1. A gas station needs to keep track of the number of gallons of gas it has on hand and the current price per gallon. The station has at least two pumps. Pumps 0 and 1 are full-service pumps, and all the rest are self-service. Self-service customers pay the base price for each gallon of gas, while full-service customers pay $0.25 more per gallon.

   Two classes are used to represent this situation. The `Pump` class handles the details for each pump and the `Station` class handles the overall gas station functions.

   Consider the following class declarations.

```
class Pump
{
  public:

    Pump();
    // postcondition: sets number of gallons sold at this pump to 0.0

    double GallonsSold() const;
    // postcondition: returns the number of gallons sold at this pump

    void ResetGallonsSold();
    // postcondition: resets number of gallons sold at this pump to 0.0

    // ... other public and private members not shown
};

class Station
{
  public:
    // constructor not shown

    double TotalSales() const;
    // postcondition: returns the total cash value of
    //                sales for all pumps

    void ResetAll();
    // postcondition: for every Pump p in this station
    //                p.GallonsSold() is 0.0

    void CloseStation(ostream & logFile);
    // precondition:  logFile is open and ready for writing
    // postcondition: writes the total cash value of
    //                all gas sold for the day to logFile;
    //                for every Pump p in this station
    //                p.GallonsSold() is 0.0

    // ... other public member functions not shown

  private:
    double myBasePrice;      // current price per gallon of gas
                             // for self-service pumps

    apvector<Pump> myPumps; // the gas pumps; myPumps.length() > 1 and
                             // is the number of pumps in this station

    // ... other private data not shown
};
```

GO ON TO THE NEXT PAGE.

(a) Write the `Station` member function `ResetAll`, as started below. `ResetAll` changes the number of gallons sold at each pump to 0.0.

In writing `ResetAll`, you may call any of the public member functions of the `Pump` and `Station` classes. Assume that all these functions work as specified.

Complete function `ResetAll` below.

```
void Station::ResetAll()
// postcondition: For every Pump p in this station
//                p.GallonsSold() is 0.0
```

(b) Write the `Station` member function `TotalSales`, as started below. `TotalSales` returns the total cash value of the gallons sold at all pumps. Recall that self-service pumps charge the base price for each gallon of gas, while full-service pumps (pumps 0 and 1) charge $0.25 more per gallon.

In writing `TotalSales`, you may call any of the public member functions of the `Pump` and `Station` classes. Assume that all these functions work as specified.

Complete function `TotalSales` below.

```
double Station::TotalSales() const
// postcondition: returns the total cash value of sales
//                for all pumps
```

(c) Write the `Station` member function `CloseStation`, as started below. `CloseStation` will write the total amount of money earned from the day's gas sales to the output stream, `logFile`, and then reset the number of gallons sold at each individual pump to 0.0.

In writing `CloseStation`, you may call any of the public member functions of the `Pump` and `Station` classes. Assume that these functions, including `ResetAll` and `TotalSales`, work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `CloseStation` below.

```
void Station::CloseStation(ostream & logFile)
// precondition:  logFile is open and ready for writing
// postcondition: writes the total cash value of
//                all gas sold for the day to logFile;
//                for every Pump p in this station
//                p.GallonsSold() is 0.0
```

GO ON TO THE NEXT PAGE.

2. Consider the following declarations for maintaining a list of books. Information about each book includes the title, author, and an appropriate age range for readers. The list is ordered by age range, as defined by function `LessThan`. Assume that a book appears at most once in the list.

```
struct Book
{
  apstring title;  // title of book
  apstring author; // author of book
  int lowAge;      // lowest recommended age
  int highAge;     // highest recommended age
};

bool LessThan(const Book & lhs, const Book & rhs);
// postcondition: returns true if lowAge of lhs < lowAge of rhs or
//                if lowAge of lhs and rhs are equal
//                  and highAge of lhs < highAge of rhs;
//                otherwise, returns false

class BookList
{
  public:

    BookList();    // constructor

    void InsertOne(const Book & bk);
    // precondition:  this BookList is in sorted order by age range
    //                as defined by LessThan;
    //                bk is not already in this BookList
    // postcondition: bk has been inserted into this BookList,
    //                maintaining its order by age range

    void InsertMany(const apvector<Book> & second);
    // precondition:  this BookList is in sorted order by age range
    //                as defined by LessThan; second contains
    //                second.length() books in arbitrary order;
    //                none of the books in second are in this BookList
    // postcondition: all the books from second have been inserted into
    //                this BookList, maintaining its order by age range

    // ... other public member functions not shown

  private:

    apvector<Book> myList;
       // collection of books in sorted order as defined by LessThan;
       // myList.length() > 0

    int myCount;
       // number of books in myList
};
```

GO ON TO THE NEXT PAGE.

(a) Write the free function `LessThan`, as started below. `LessThan` returns `true` if either

- `lowAge` of the first book is less than `lowAge` of the second book; or
- `lowAge` is the same for both books, and `highAge` of the first book is less than `highAge` of second book.

Otherwise, `LessThan` returns `false`.

For example:

BookA		BookB		LessThan(BookA, BookB)
lowAge	highAge	lowAge	highAge	
9	12	9	14	true
9	12	10	11	true
9	12	10	15	true
9	12	8	15	false
9	12	9	11	false
9	12	9	12	false

Complete function `LessThan` below.

```
bool LessThan(const Book & lhs, const Book & rhs)
// postcondition: returns true if lowAge of lhs < lowAge of rhs or
//                if lowAge of lhs and rhs are equal
//                  and highAge of lhs < highAge of rhs;
//                otherwise, returns false
```

GO ON TO THE NEXT PAGE.

(b) Write the `BookList` member function `InsertOne`, as started below. Assume that the private data member `myList` is already ordered as defined by `LessThan`. `InsertOne` places a book into the `BookList`, maintaining that order, and will resize `myList` by doubling the capacity, if necessary.

For example, assume that `BookList library` contains the following books.

title	author	lowAge	highAge
Madeline	Bemelmans	3	8
The Lorax	Seuss	3	10
Harry Potter and the Sorcerer's Stone	Rowling	9	99
Holes	Sacher	12	18
I Know Why the Caged Bird Sings	Angelou	16	99

Consider the following `Book bk` to be inserted into `library`.

title	author	lowAge	highAge
Little House on the Prairie	Wilder	8	12

After the call `library.InsertOne(bk)`, `library` contains the following books. Note that the books are in sorted order by `lowAge`, then by `highAge` within `lowAge`.

title	author	lowAge	highAge
Madeline	Bemelmans	3	8
The Lorax	Seuss	3	10
Little House on the Prairie	Wilder	8	12
Harry Potter and the Sorcerer's Stone	Rowling	9	99
Holes	Sacher	12	18
I Know Why the Caged Bird Sings	Angelou	16	99

In writing `InsertOne`, you may call function `LessThan` specified in part (a). Assume that `LessThan` works as specified, regardless of what you wrote in part (a).

Complete function `InsertOne` below.

```
void BookList::InsertOne(const Book & bk)
// precondition:  this BookList is in sorted order by age range
//                as defined by LessThan;
//                bk is not already in this BookList
// postcondition: bk has been inserted into this BookList,
//                maintaining its order by age range
```

GO ON TO THE NEXT PAGE.

(c) Write the `BookList` member function `InsertMany`, as started below. `InsertMany` will insert all the books from an array of books into this `BookList`, maintaining the sorted order of the private data member `myList`, as defined by `LessThan`.

For example, assume that the array `books` contains the following list of books to be inserted into the initial version of `library` shown in part (b).

	title	**author**	**lowAge**	**highAge**
0	The Phantom Tollbooth	Juster	9	12
1	Invisible Man	Ellison	15	99
2	Charlotte's Web	White	8	12

The following table shows the contents of `library` after the call `library.InsertMany(books)`.

title	**author**	**lowAge**	**highAge**
Madeline	Bemelmans	3	8
The Lorax	Seuss	3	10
Charlotte's Web	White	8	12
The Phantom Tollbooth	Juster	9	12
Harry Potter and the Sorcerer's Stone	Rowling	9	99
Holes	Sacher	12	18
Invisible Man	Ellison	15	99
I Know Why the Caged Bird Sings	Angelou	16	99

In writing `InsertMany`, you may call functions `LessThan` and `InsertOne` specified in parts (a) and (b). Assume that `LessThan` and `InsertOne` work as specified, regardless of what you wrote in parts (a) and (b). Do not assume that the list of books to be added is in any particular order.

Complete function `InsertMany` below.

```
void BookList::InsertMany(const apvector<Book> & second)
// precondition:  this BookList is in sorted order by age range
//                as defined by LessThan; second contains
//                second.length() books in arbitrary order;
//                none of the books in second are in this BookList
// postcondition: all the books from second have been inserted into
//                this BookList, maintaining its order by age range
```

GO ON TO THE NEXT PAGE.

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

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

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

GO ON TO THE NEXT PAGE.

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

```
class Fish
{
  public:
    // constructors

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

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

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

    // accessing functions

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

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

    // modifying functions

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

  private:

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

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

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

    int myId;
    Position myPos;
    bool amIDefined;

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

GO ON TO THE NEXT PAGE.

The `Environment` class will have the following changes.

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

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

```
class Environment
{
  public:
    // constructor

    Environment(istream & input);

    // accessing functions

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

    // modifying functions

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

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

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

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

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

GO ON TO THE NEXT PAGE.

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

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

Complete function `RemoveFish` below.

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

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

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

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

Complete function `Breed` below.

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

GO ON TO THE NEXT PAGE.

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

Note: If `r` is defined as follows,

```
RandGen r;
```

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

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

Complete function `Act` below.

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

GO ON TO THE NEXT PAGE.

4. A window is represented by an *M*-by-*N* matrix filled with integers representing colors. Operations on a window include the following.

- Determine if a point lies within the window.
- Place a square of a single color in the window, ignoring those points in the square that are not within the window.

Consider the following declarations for `Window`.

```
class Window
{
  public:

    // ... constructors not shown

    bool IsInBounds(int row, int col) const;
    // postcondition: returns true if the point (row, col) is
    //                in this window;
    //                otherwise, returns false

    void ColorSquare(int ULrow, int ULcol, int N, int val);
    // postcondition: all points in this window that are also in the
    //                N-by-N square with upper left corner
    //                (ULrow, ULcol) have been set to val;
    //                points in the square that are not in this
    //                window are ignored

    int ValAt(int row, int col) const;
    // postcondition: returns color value at position row, col
    //                in this window

    // ... other public member functions not shown

  private:
    int myNumRows;
    int myNumCols;
    apmatrix<int> myMat;
};
```

(a) Write the `Window` member function `IsInBounds`, as started below. `IsInBounds` checks whether a single point is in the window.

For example, for any 5-by-4 `Window W`, the following table shows the results of several calls to `IsInBounds`.

Call	Return value
`W.IsInBounds(0, 0)`	`true`
`W.IsInBounds(2, 1)`	`true`
`W.IsInBounds(4, 3)`	`true`
`W.IsInBounds(5, 3)`	`false`
`W.IsInBounds(3, -1)`	`false`
`W.IsInBounds(8, 8)`	`false`

GO ON TO THE NEXT PAGE.

Complete function `IsInBounds` below.

```
bool Window::IsInBounds(int row, int col) const
// postcondition: returns true if the point (row, col) is
//                in this window;
//                otherwise, returns false
```

(b) Write the `Window` member function `ColorSquare`, as started below. `ColorSquare` sets all the integers in a specified square to a particular color value. `ULrow` and `ULcol` specify the location of the upper left corner of the square, `N` is the number of rows and columns in the square, and `val` is the color value. Points that are in the specified square but do not lie in the `Window` are ignored.

For example, consider the following 5-by-6 `Window W`.

```
10 10 10 10 10 10
10 10 10 20 20 20
20 20 30 30 30 30
30 30 40 40 40 40
40 40 50 50 50 50
```

After the call `W.ColorSquare(2, 1, 3, 66)`, `W` is changed to

```
10 10 10 10 10 10
10 10 10 20 20 20
20 66 66 66 30 30
30 66 66 66 40 40
40 66 66 66 50 50
```

After an additional call, `W.ColorSquare(2, 4, 3, 77)`, `W` is changed to

```
10 10 10 10 10 10
10 10 10 20 20 20
20 66 66 66 77 77
30 66 66 66 77 77
40 66 66 66 77 77
```

Note that the third column of the square added is not in `W`.

In writing function `ColorSquare`, you may call function `IsInBounds` specified in part (a). Assume that `IsInBounds` works as specified, regardless of what you wrote in part (a).

Complete function `ColorSquare` below.

```
void Window::ColorSquare(int ULrow, int ULcol, int N, int val)
// postcondition: all points in this window that are also in the
//                N-by-N square with upper left corner
//                (ULrow, ULcol) have been set to val;
//                points in the square that are not in this
//                window are ignored
```

GO ON TO THE NEXT PAGE.

(c) A rectangular area in a window can be specified using the `Rectangle` structure as declared below.

```
struct Rectangle
{
  int ULrow;    // row position of upper left corner of rectangle
  int ULcol;    // column position of upper left corner of rectangle
  int numRows; // number of rows in rectangle (height)
  int numCols; // number of columns in rectangle (width)
};
```

The following example shows a 5-by-4 window in which the 3-by-2 rectangle with upper left corner (2,1) is highlighted.

10	20	30	40
10	20	30	40
10	**99**	**55**	40
10	**44**	**33**	40
10	**77**	**66**	40

Write the free function `Enlarge`, as started below. `Enlarge` magnifies a `Rectangle` in the `Window` by replacing each point with a `factor-by-factor` square of points of the same color. The upper left corner of the magnified `Rectangle` is the same as the upper left corner of the original `Rectangle`. Each square of color is placed in the `Window` at the same relative position in the magnified `Rectangle` as the original point in the `Rectangle`. Conceptually, the enlarged rectangle may run off the window, but only points in the window are modified by `Enlarge`.

For example, consider the 10-by-11 `Window W`, and `Rectangle R`, where `R.ULrow = 2`, `R.ULcol = 1`, `R.numRows = 2`, and `R.numCols = 4`. The following table shows the original version of `W` with `R` highlighted and the result of magnifying `R` in `W` by a `factor` of `3`.

<u>`Window W` with `R` highlighted</u>

00	00	00	10	10	10	10	10	10	10	10
10	10	10	20	20	20	20	20	20	20	20
20	**55**	**99**	**33**	**66**	20	20	20	30	30	30
30	**22**	**88**	**77**	**44**	30	30	30	40	40	40
40	40	40	40	30	30	30	50	50	50	50
50	50	50	40	40	40	40	30	30	30	30
60	60	50	50	50	40	40	40	30	30	20
70	70	70	50	50	50	40	40	40	30	30
80	80	70	70	60	60	50	50	40	40	30
90	80	70	60	60	50	50	40	40	30	20

<u>Result of the call `Enlarge(W, R, 3)`</u>

00	00	00	10	10	10	10	10	10	10	10
10	10	10	20	20	20	20	20	20	20	20
20	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
30	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
40	**55**	**55**	**55**	**99**	**99**	**99**	**33**	**33**	**33**	**66**
50	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
60	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
70	**22**	**22**	**22**	**88**	**88**	**88**	**77**	**77**	**77**	**44**
80	80	70	70	60	60	50	50	40	40	30
90	80	70	60	60	50	50	40	40	30	20

In writing `Enlarge`, you may call any public `Window` member functions. Assume that `IsInBounds` and `ColorSquare` work as intended, regardless of what you wrote in parts (a) and (b).

Complete function `Enlarge` below.

```
void Enlarge(Window & W, const Rectangle & rect, int factor)
// precondition:  factor > 0
```

END OF EXAMINATION