AP® Computer Science A
2003 Free-Response Questions

These materials were produced by Educational Testing Service® (ETS®), which develops and administers the examinations of the Advanced Placement Program for the College Board. The College Board and Educational Testing Service (ETS) are dedicated to the principle of equal opportunity, and their programs, services, and employment policies are guided by that principle.

The College Board is a national nonprofit membership association whose mission is to prepare, inspire, and connect students to college and opportunity. Founded in 1900, the association is composed of more than 4,300 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 22,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of equity and excellence, and that commitment is embodied in all of its programs, services, activities, and concerns.

For further information, visit www.collegeboard.com

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.

1. Information about a `College` object includes its name, its tuition, and the region in which it is located.

The following class declaration will be used to store information about a college.

```
class College
{
  public:
    apstring Name() const;
    // returns college name

    apstring Region() const;
    // returns region of college

    int Tuition() const;
    // returns college tuition

    void SetTuition(int newTuition);
    // set college's tuition to newTuition

  // ... constructors, other member functions and data not shown
};
```

The following class declaration will be used to store information about a group of colleges.

```
class CollegeGroup
{
  public:

    void UpdateTuition(const apstring & collegeName,
                       int newTuition);
    // precondition:  collegeName exists in this CollegeGroup
    // postcondition: tuition for collegeName is changed to newTuition

    apvector<College> GetCollegeList(const apstring & region,
                                     int low, int high) const;
    // precondition:  low <= high
    // postcondition: returns array of colleges in region
    //                where low <= tuition <= high;
    //                the size of the array returned is equal to the
    //                number of colleges that meet the criteria


  private:

    apvector<College> myColleges;
       // myColleges.length() is the number of colleges

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

The following chart shows an example of colleges that could appear in an object of type `CollegeGroup`.

	Name	Region	Tuition
0	Colgate University	Northeast	$27,025
1	Duke University	Southeast	$26,000
2	Kalamazoo College	Midwest	$19,764
3	Stanford University	West	$25,917
4	Florida International University	Southeast	$10,800
5	Dartmouth College	Northeast	$27,764
6	Spelman College	Southeast	$11,455

(a) Write the `CollegeGroup` member function `UpdateTuition`, which is described as follows. `UpdateTuition` changes the tuition of the college whose name is passed as a parameter.

For example, if the object `colleges` is of type `CollegeGroup` and contains the entries shown in the chart above, the call `colleges.UpdateTuition("Colgate University", 27500)` would change the tuition of Colgate University to $27,500.

Complete function `UpdateTuition` below.

```
void CollegeGroup::UpdateTuition(const apstring & collegeName,
                                 int newTuition)
// precondition:  collegeName exists in this CollegeGroup
// postcondition: the tuition for collegeName is changed to newTuition
```

(b) The table below is repeated for your convenience.

	Name	Region	Tuition
0	Colgate University	Northeast	$27,025
1	Duke University	Southeast	$26,000
2	Kalamazoo College	Midwest	$19,764
3	Stanford University	West	$25,917
4	Florida International University	Southeast	$10,800
5	Dartmouth College	Northeast	$27,764
6	Spelman College	Southeast	$11,455

Write the `CollegeGroup` member function `GetCollegeList`, which is described as follows. `GetCollegeList` returns an array of colleges that are located in the specified region and whose tuition is in the range between `low` and `high`, inclusive. The size of the array should be equal to the number of colleges that meet the criteria of region and tuition range.

For example, if the object `colleges` is of type `CollegeGroup` and contains the entries shown in the chart above, the call `colleges.GetCollegeList("Southeast", 10000, 20000)` should return an array of size two containing Florida International University and Spelman College (note that Duke University is not included because its tuition is not in the specified range and Kalamazoo College is not included because it is not in the specified region).

Complete function `GetCollegeList` below.

```
apvector<College> CollegeGroup::GetCollegeList(const apstring & region,
                                                int low, int high) const
// precondition:  low <= high
// postcondition: returns array of colleges in region
//                where low <= tuition <= high;
//                the size of the array returned is equal to the number
//                of colleges that meet the criteria
```

2. Periodically, a company processes the retirement of some of its employees. In this question, you will write functions to help the company determine whether an employee is eligible to retire and to process the retirement of all eligible employees.

The `Employee` class is declared as follows.

```
class Employee
{
  public:
    int Age() const;
    // returns the age (in years) of this employee

    int YearsOnJob() const;
    // returns the number of years this employee has worked

    double Salary() const;
    // returns the salary of this employee in dollars

    int ID() const;
    // returns unique employee ID number

  // ... constructors, other member functions and data not shown
};
```

The `Company` class is declared as follows.

```
class Company
{
  public:
    void ProcessRetirements();
    // postcondition: all retirement-eligible employees have been
    //                removed from empList; empList has been resized
    //                to reflect retirements;
    //                empList remains sorted by employee ID;
    //                salaryBudget has been updated to reflect remaining
    //                employees

    // ... constructor and other public methods not shown

  private:
    bool EmployeeIsEligible(const Employee & emp) const;
    // postcondition: returns true if emp is eligible to retire;
    //                otherwise, returns false

    apvector<Employee> empList;
    // empList.length() is the number of employees in this company

    int retireAge;           // minimum age to retire
    int retireYears;         // minimum years on job to retire
    double retireSalary;     // minimum salary to retire

    double salaryBudget;
    // total salary of all employees

};
```

The data member `empList` is sorted in ascending order by employee ID. The total of all salaries is maintained in the data member `salaryBudget`.

(a) An employee is eligible for retirement if (s)he meets at least two of the following requirements:

1. The employee is at least `retireAge` years old.
2. The employee has worked for at least `retireYears`.
3. The employee's salary is at least `retireSalary`.

Write the `Company` member function `EmployeeIsEligible`, which is described as follows. `EmployeeIsEligible` returns a Boolean value that indicates whether `Employee emp` is eligible for retirement, using the rules described above.

Complete function `EmployeeIsEligible` below.

```
bool Company::EmployeeIsEligible(const Employee & emp) const
// postcondition: returns true if emp is eligible to retire;
//                otherwise, returns false
```

(b) Write the `Company` member function `ProcessRetirements`, which is described as follows. `ProcessRetirements` removes all retirement-eligible employees from the `empList` array, resizes (shrinks) `empList` as appropriate (maintaining its order by employee ID), and decreases `salaryBudget` to reflect the salary of the remaining employees.

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

Complete function `ProcessRetirements` below.

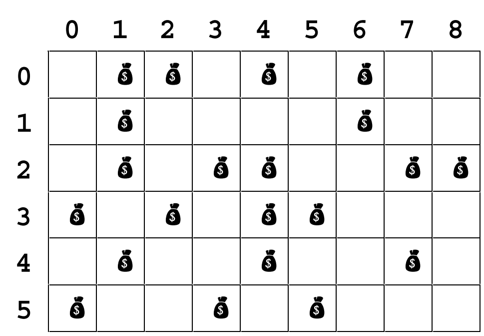
```
void Company::ProcessRetirements()
// postcondition: all retirement-eligible employees have been
//                removed from empList; empList has been resized
//                to reflect retirements;
//                empList remains sorted by employee ID;
//                salaryBudget has been updated to reflect remaining
//                employees
```

3. A treasure map is represented as a rectangular grid. Each grid location contains either a single treasure or nothing. The grid is represented using a matrix of Boolean values. If a cell in the grid contains a treasure then the value `true` is stored in the corresponding matrix location; otherwise, the value `false` is stored.

Consider the following declaration for the `TreasureMap` class.

```
class TreasureMap
{
  public:

    // ... constructors not shown

    bool HasTreasure(int row, int col) const;
    // postcondition: returns true if the cell at location (row, col)
    //                contains a treasure;
    //                returns false if location (row, col) is not within
    //                the bounds of the grid or if there is no treasure
    //                at that location

    int NumAdjacent(int row, int col) const;
    // precondition:  0 <= row < NumRows(); 0 <= col < NumCols()
    // postcondition: returns a count of the number of treasures in the
    //                cells adjacent to the location (row, col),
    //                horizontally, vertically, and diagonally

    int NumRows() const;
    // postcondition: returns the number of rows in the treasure map

    int NumCols() const;
    // postcondition: returns the number of columns in the treasure map

  private:

    apmatrix<bool> myGrid;
       // myGrid[r][c] being true indicates a treasure at (r, c)
       // the matrix is sized by the constructor
};
```

For example, suppose that the 6-by-9 grid shown below is a treasure map where the symbol in a cell indicates a treasure. In this example, `myGrid[2][3]` is `true` and `myGrid[1][2]` is `false`.



GO ON TO THE NEXT PAGE.

(a) Write the `TreasureMap` member function `HasTreasure`, which is described as follows. `HasTreasure` returns `true` if there is a treasure at the location `(row, col)`. If `(row, col)` is not within the bounds of the grid or if there is no treasure at that location, `HasTreasure` returns `false`.

For example, if `TreasureMap theMap` represents the treasure map shown at the beginning of the question, the following table gives the results of several calls to `HasTreasure`.

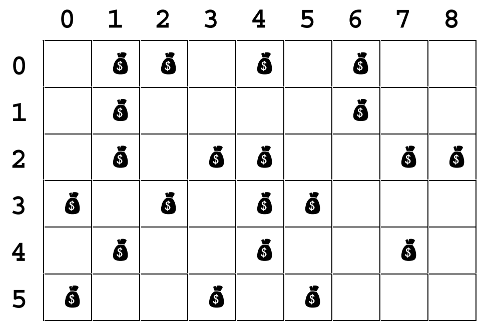
Function call	Value returned
`theMap.HasTreasure(0, 2)`	`true`
`theMap.HasTreasure(0, -1)`	`false`
`theMap.HasTreasure(2, 3)`	`true`
`theMap.HasTreasure(2, 2)`	`false`
`theMap.HasTreasure(4, 9)`	`false`

Complete function `HasTreasure` below.

```
bool TreasureMap::HasTreasure(int row, int col) const
// postcondition: returns true if the cell at location (row, col)
//                contains a treasure;
//                returns false if location (row, col) is not within
//                the bounds of the grid or if there is no treasure
//                at that location
```

(b) Write the `TreasureMap` member function `NumAdjacent`, which is described as follows. `NumAdjacent` returns the number of treasures that are adjacent to a given location specified by `row` and `col`. To be adjacent, a treasure must be in one of the (at most) eight cells that border the given location horizontally, vertically, or diagonally; a treasure in the given location does not count as being adjacent.

The treasure map below is repeated for your convenience.



For example, if `TreasureMap theMap` represents the treasure map shown above, the following table gives the results of several calls to `NumAdjacent`.

Function call	Value returned
`theMap.NumAdjacent(3, 3)`	5
`theMap.NumAdjacent(2, 4)`	3
`theMap.NumAdjacent(4, 7)`	0

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

Complete function `NumAdjacent` below.

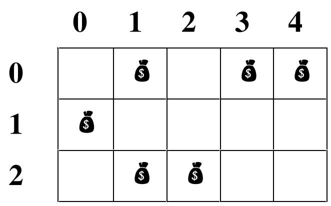
```
int TreasureMap::NumAdjacent(int row, int col) const
// precondition:  0 <= row < NumRows(); 0 <= col < NumCols()
// postcondition: returns a count of the number of treasures in the
//                cells adjacent to the location (row, col),
//                horizontally, vertically, and diagonally
```


GO ON TO THE NEXT PAGE.

(c) Write free function `ComputeCounts`, which is described as follows. `ComputeCounts` returns a matrix of integers where the value at `(row, col)` is `9` if there is a treasure at location `(row, col)` in `theMap`. Otherwise, the value at `(row, col)` is the number of treasures adjacent to location `(row, col)`.

For example, the following shows the matrix that is returned as a result of calling `ComputeCounts` with the `TreasureMap aMap`.

<u>`aMap`</u>



<u>Matrix returned by the call `ComputeCounts(aMap)`</u>

	0	1	2	3	4
0	2	9	2	9	9
1	9	4	4	3	2
2	2	9	9	1	0

In writing `ComputeCounts`, you may call any `TreasureMap` member function. Assume that all member functions of `TreasureMap` work as specified, regardless of what you wrote in parts (a) and (b).

Complete function `ComputeCounts` below.

```
apmatrix<int> ComputeCounts(const TreasureMap & theMap)
```

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

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

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

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

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

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

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


//  Added to the private section of class Environment

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

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

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

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


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

    void Move(Environment & env);


//  Added to the private section of class Fish

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

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

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

Complete function `NumAlgaeAt` below.

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

(b) Write the `Fish` member function `MostAlgae`, which is described as follows. `MostAlgae` should return a `Position` from `nbrs` that contains the most algae. If more than one position contains the maximum amount, any of those positions may be returned.

In writing `MostAlgae`, you may use any of the `Environment` public member functions, including `NumAlgaeAt`. Assume that `NumAlgaeAt` works as specified, regardless of what you wrote in part (a).

Complete function `MostAlgae` below.

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

(c) Write the `Fish` member function `Act`, which is described as follows. If there is algae at the fish's current position, the fish should eat one unit of algae and not move. If there is no algae and this is the third consecutive step in which the fish has not eaten, `Act` will cause the fish to die by calling `env.RemoveFish`. If the fish does not eat and does not die, then the fish should move to a neighboring position with the most algae. `Act` should update the state of the environment and the state of the fish appropriately.

In writing `Act`, you may use any member function from the Marine Biology Case Study, including those added at the beginning of the question. Assume that `Fish::Move` has been modified to work correctly and that `Environment::NumAlgaeAt` and `Fish::MostAlgae` 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: if there was algae at Location(), this Fish ate
//                and one unit of algae has been removed from
//                Location(); otherwise, if this was the third
//                consecutive step that this Fish did not eat,
//                then this Fish has been removed from env;
//                otherwise, this Fish moved.
//                myStepsSinceFed has been updated.
```

END OF EXAMINATION