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`.

	0	1	2	3	4	5	6	7	8
0		💰	💰		💰		💰		
1		💰					💰		
2		💰		💰	💰			💰	💰
3	💰		💰		💰	💰			
4		💰			💰			💰	
5	💰			💰		💰			

AP® COMPUTER SCIENCE A
2003 SCORING GUIDELINES

Question 2

Part A: `EmployeeIsEligible` **3 pts**

Using count method

+2 count conditions satisfied
- **+1** attempt (at least two conditions checked, possible error in comparison)
- **+1** correct (count value is correct for all cases)

+1 return correct value relative to the count

Using Boolean expression

+1 attempt (at least two conditions checked, possible error in comparison)

+1 partial (correct value determined for at least two different true cases)
(must have some attempt to identify a pair) (must have an `emp.`)

+1 all cases correct (except for `==`, `>=` discrepancies)

Part B: `ProcessRetirements` **6 pts**

+1 check eligible
- **+1/2** attempt (must have reference to an `Employee`)
- **+1/2** correct

+1 loop over `empList`
- **+1/2** attempt (must have context of eligible check)
- **+1/2** correct (watch for fixed loop bounds such as len)

+1 1/2 place non-eligible into correct position
- **+1/2** attempt
- **+1** correct (extra vector method must have declaration with proper size and copy back)
(index/count for placement of non-eligible must be correctly implemented)
(lose this point if shift array method skips consecutive eligible employees)

+1 adjust salary budget to account for retirements
- **+1/2** correctly computed
- **+1/2** stored in `salaryBudget` instance variable

+1 `empList` has been resized correctly

+1/2 initialize count of non-eligible employees or index for placing non-eligible employees
(get this if there is an attempt to shift multiple array elements for each employee removed)