Seat assignments are processed by the public member functions of the class `Flight`. The seating arrangement is represented internally by a matrix of seats in the class `Flight`. The declaration for the class `Flight` is as follows.

```
class Flight
{
  public:
    int EmptySeatCount(const apstring & seatType) const;
    // postcondition: returns the number of empty seats
    //                whose type is seatType;
    //                if seatType is "any", returns the
    //                total number of empty seats

    int FindBlock(int row, int seatsNeeded) const;
    // postcondition: returns column index of the first (lowest index)
    //                seat in a block of seatsNeeded adjacent
    //                empty seats in the specified row;
    //                if no such block exists, returns -1

    bool AssignGroup(const apvector<Passenger> & group);
    // postcondition: if possible, assigns the group.length() passengers
    //                from group to adjacent empty seats in a single row
    //                and returns true;
    //                otherwise, makes no changes and returns false

    // ... constructors and other public member functions not shown

  private:
    apmatrix<Seat> mySeats;

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

(a) You will write the `Flight` member function `EmptySeatCount`, which is described as follows. `EmptySeatCount` returns the number of empty seats of the specified type `seatType`. Recall that an empty seat holds a default passenger whose name is `""`. If `seatType` is `"any"`, then every empty seat should be counted in determining the number of empty seats. Otherwise, only seats whose type is the same as `seatType` are counted in determining the number of empty seats.

For example, consider the diagram of passengers assigned to seats as stored in `mySeats` for `Flight ap2002` as shown below.

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	window "Kelly"	middle "Robin"	aisle ""	aisle "Sandy"	middle ""	window "Fran"
[1]	window "Chris"	middle "Alex"	aisle ""	aisle ""	middle "Pat"	window "Sam"

The following table shows several examples of calling `EmptySeatCount` for this flight.

Function Call	Value Returned
`ap2002.EmptySeatCount("aisle")`	`3`
`ap2002.EmptySeatCount("window")`	`0`
`ap2002.EmptySeatCount("middle")`	`1`
`ap2002.EmptySeatCount("any")`	`4`

Complete function `EmptySeatCount` below.

```
int Flight::EmptySeatCount(const apstring & seatType) const
// postcondition: returns the number of empty seats
//                whose type is seatType;
//                if seatType is "any", returns the
//                total number of empty seats
```

AP® COMPUTER SCIENCE A
2002 SCORING GUIDELINES

Question 4

Part A: EmptySeatCount **3 points**

+1 Loop over matrix
- **+1/2** attempt (traverse and index some matrix in two dimensions: *anything*[*var1*][*var2*] shows access to multiple rows with multiple columns in each row)
- **+1/2** correct

+1 1/2 Identify empty seats of correct type
- **+1/2** attempt (*EmptyTestAttempt** OR compares seatType against something)
- **+1/2** correct use of abstraction
- **+1/2** correctly identify all empty seats

+1/2 Count – initialize counter, conditionally increment counter, return value

Note: If specific columns are used for seat types, cannot get any Identify points.

Part B: FindBlock **3 points**

Note: No deduction for missing check of parameters; bad check can lose Traverse correct ½ point.

+1 Traverse row and test for empty seat
- **+1/2** attempt (traverse a row, compare something against empty string)
- **+1/2** correct (no out-of-bounds, correct use of abstraction)

+1 1/2 Find a block of empty seats
- **+1/2** attempt (nested traversal with *EmptyTestAttempt** OR attempt to count adjacent empty seats)
- **+1/2** traverse potential blocks (checks range of block)
- **+1/2** correctly identify & maintain block location (block with enough empty seats found if it exists)

+1/2 Return correct value (leftmost location of empty block or –1)

**EmptyTestAttempt* = compare mySeat(s) or GetPassenger(s) or GetName(s) against empty string
OR compare GetPassenger(s) with Passenger()

Question 4 (cont'd.)

Part C: AssignGroup **3 points**

Reminder: assume FindBlock returns –1 on parameters outside bounds

+1/2 Loop through rows correctly and terminate

+1 Find block of empty seats
- **+1/2** attempt (call FindBlock, use returned value, parameters optional OR reimplement correctly)
- **+1/2** correct (correct call to FindBlock OR reimplemented correctly)

+1 Assign passengers to seats if found
- **+1/2** attempt (must attempt to assign a passenger from group in context of search for block, index not required)
- **+1/2** correct

+1/2 Return correct boolean

Notes: If group is placed more than once, must lose Loop and Assign correct points.
No loop loses Loop, Assign correct, Return correct.
Constant loop loses Loop.

**EmptyTestAttempt* = compare mySeat(s) or GetPassenger(s) or GetName(s) against empty string
OR compare GetPassenger(s) with Passenger()

AP® COMPUTER SCIENCE A
2002 SCORING GUIDELINES

Grading Guidelines for AP Computer Science Free-Response Questions

The grading for each question is based on a rubric that has been developed by the question and exam leaders. The rubric allocates points to different elements of a solution.

A common pattern for a rubric is to indicate a point (or half point) for an attempt at an element of a solution, with another point if that element is correct. In general, the attempt point is given if there is clear evidence that the student understands that element of the problem. For example, a loop that is clearly an attempt to iterate over the correct range would get the attempt point but might lose the correct point if there was an off-by-one error or an incorrect increment. Similarly, an assignment or conditional that involved an expression including an array access would get an attempt point if the expression was substantially correct, but had an error with the array index or a minor error in the expression. The rubric describes what constitutes enough to warrant giving an attempt point. A correct point means correct, with the exceptions noted below.

Some elements of a problem may simply be all-or-nothing for a particular point.

The "General Usage" sheet specifies how errors not incorporated into the rubric should be handled. Some items on the "General Usage" sheet are also addressed in a rubric; in such a case, the rubric takes precedence. (Check with question leader for application of General Usage.)

Sometimes a rubric does not cover an error appropriately. For example, there might be ½ point allocated for a function to have a correct return statement. However, a missing return combined with output to the screen of the return value indicates a fundamental misconception about functions that warrants a larger penalty. On the other hand, a solution might be correct except for a minor error or an error that is repeated several times. Some minor errors should not be deducted at all and some should not be deducted repeatedly if the code is otherwise correct. The "General Usage" sheet covers these situations.

General Usage specifies certain minor errors that should not be deducted, such as missing semicolons or the use of "[r, c]" instead of "[r][c]" for accessing an apmatrix. An element of a solution with such an error can get credit for being correct.

General Usage also indicates minor errors worth ½ point and major errors worth 1 point. These errors should be taken only once on a particular problem, and not repeatedly. If a usage deduction is made, then an element of the problem can get credit for being correct on that part if it has no other errors.

Usage points cannot be deducted for a part of a problem that would thereby receive a negative score.

AP® COMPUTER SCIENCE A
2002 SCORING GUIDELINES

2002 General Usage

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

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

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

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

Non-penalized errors

case discrepancies, unless confuses identifiers

missing ;'s

missing { }'s where indentation clearly conveys intent

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

obj.Func instead of obj.Func( )

loop variables used outside loop

[r, c] instead of [r][c]

= instead of == (and vice-versa)

missing ( )'s around if/while tests

<< instead of >> (and vice-versa)

*foo.data instead of (*foo).data

Minor errors (1/2 point)

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

no variables declared

void function returns a value

modifying a const parameter

unnecessary cout << "done"

unnecessary cin (to pause)

no * in pointer declaration

use of L->item when L.item is correct

Major errors (1 point)

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

function result written to output

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

MemberFunction(obj) instead of obj.MemberFunction( )

param.FreeFunction( ) instead of FreeFunction(param)

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

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

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