AP® Computer Science A
2002 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 dedicated to preparing, inspiring, and connecting students to college and opportunity. Founded in 1900, the association is composed of more than 4,200 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 admission, 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.

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. A researcher wishes to calculate some statistical properties for a collection of integer data values. The data values are represented by the array `tally`. The indexes of the array represent the possible values of the actual data values from zero to the maximal value (15 in the example below). Each array location contains the frequency (number of occurrences) of the value corresponding to its index. In the example below, `tally[4]` is 10, which means that the value **4** occurs ten times in the collection of data; whereas `tally[8]` is 0, which means that the value **8** does not occur in the data collection.

`tally`

Value	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Frequency	0	0	10	5	10	0	7	1	0	6	0	10	3	0	0	1

(a) You will write the function `CalculateModes`, which is described as follows. `CalculateModes` returns an array containing the mode(s) found in parameter `tally`. The length of the returned array is equal to the number of modes.

A **mode** is defined as a value that occurs with maximal frequency. If there is more than one such value, each is considered a mode of the data. In the example above, the modes are 2, 4, and 11, because they each occur 10 times and all other values occur fewer than 10 times.

The following function, `FindMax`, is available for your use. It returns the maximum value in array `nums`. Using the example array, `FindMax(tally)` returns 10.

```
int FindMax(const apvector<int> & nums);
// precondition:  nums.length() > 0
// postcondition: returns the maximum value in nums
```

Do NOT write the body of `FindMax`.

In writing `CalculateModes`, you may call `FindMax` as specified above.

Complete function `CalculateModes` below.

```
apvector<int> CalculateModes(const apvector<int> & tally)
// precondition:  tally.length() > 0
// postcondition: returns an apvector that contains the mode(s);
//                the apvector's length equals the number of modes
```

(b) You will write the function `KthDataValue`, which is described as follows. `KthDataValue` returns the kth data value when the data values are considered in sorted order. Recall that the indexes of the array represent possible data values and that each array location contains the frequency of the value corresponding to its index.

In the example reprinted below, the first ten data values are **2**, the next five data values are **3**, and the next ten data values are **4**. `KthDataValue(tally, 1)` returns **2**, `KthDataValue(tally, 14)` returns **3**, `KthDataValue(tally, 15)` returns **3**, and `KthDataValue(tally, 16)` returns **4**.

`tally`

Value	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Frequency	0	0	10	5	10	0	7	1	0	6	0	10	3	0	0	1

Complete function `KthDataValue` below.

```
int KthDataValue(const apvector<int> & tally, int k)
// precondition:  tally.length() > 0;
//                0 < k ≤ total number of values in the data collection
// postcondition: returns the kth value in the data collection
//                represented by tally
```

2. Consider the following declaration that will be used to keep track of information about items in a grocery store . Each item is identified by a unique one-word name and has an associated price, size, and category.

```
class GroceryStore
{
  public:
    GroceryStore();

    // modifier

    void SetPrice(const apstring & itemName, double price);
       // changes the price of item associated with itemName

    // accessors

    double GetPrice(const apstring & itemName) const;
       // returns the price of this item

    int GetSize(const apstring & itemName) const;
       // returns the size (in ounces) of this item

    apvector<apstring> GetItems(char category) const;
       // returns a vector (possibly empty) of the names of all
       // items in the specified category

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

(a) You will write free function `ChangePrices`, which is described as follows. `ChangePrices` reads item names and prices from `input` and changes the prices of the corresponding items in `store` to the new prices.

For example, assume `store` contains the following items.

Name	Price	Size (in ounces)	Category
avocado	1.68	8	P
milk	1.92	64	D
chicken	4.48	64	M
broccoli	1.92	16	P
yogurt	0.96	16	D
spinach	1.76	16	P
cornedbeef	6.72	48	M
porkchops	2.24	32	M

Assume that the stream `input` contains the following data.

```
cornedbeef  7.99
yogurt       .75
milk        1.25
broccoli     .98
```

The call `ChangePrices(store, input)` will change the prices of `cornedbeef`, `yogurt`, `milk`, and `broccoli` to the corresponding new prices.

In writing `ChangePrices`, you may call any of the public member functions of the `GroceryStore` class. Assume the member functions work as specified.

Complete free function `ChangePrices` below.

```
void ChangePrices(GroceryStore & store, istream & input)
// precondition:  input is open for reading;
//                each line consists of a valid one word item name
//                and a valid price
// postcondition: changes the prices of items in store using names and
//                new prices from input
```

(b) The unit price of an item is the price per ounce. The table below is repeated from part (a) for your convenience.

Name	Price	Size (in ounces)	Category
avocado	1.68	8	P
milk	1.92	64	D
chicken	4.48	64	M
broccoli	1.92	16	P
yogurt	0.96	16	D
spinach	1.76	16	P
cornedbeef	6.72	48	M
porkchops	2.24	32	M

The unit price of avocado is 1.68 divided by 8, which equals 0.21, and the unit price of spinach is 1.76 divided by 16, which equals 0.11.

You will write free function `BargainItem`, which is described as follows. `BargainItem` returns the name of an item whose unit price is the lowest in the specified category. If there is more than one item with the lowest unit price, any one of these items may be returned. If there are no items in the category, `BargainItem` returns `"none"`.

For example, consider the items and prices listed in the table above. Using this table, the results of three calls to `BargainItem` are shown below.

Function call	Returned value
`BargainItem(store, 'P')`	`spinach`
`BargainItem(store, 'M')`	`chicken` or `porkchops`
`BargainItem(store, 'B')`	`none`

In writing `BargainItem`, you may call any of the public member functions of the `GroceryStore` class. Assume that the member functions work as specified.

Complete free function `BargainItem` below.

```
apstring BargainItem(const GroceryStore & store, char category)
// postcondition: returns the name of an item whose unit price
//                is the lowest in the specified category;
//                if no items in the specified category, returns "none"
```

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 fish so they have a direction based on their last move, and move only in a forward direction: either straight ahead or diagonally ahead to the right or left. You will be asked to write three functions for this question:

   (a) a `Position` member function that returns the position to the northeast of the current position,

   (b) a `Fish` member function that returns a neighborhood of positions representing the forward moves that a fish can make, and

   (c) a `Position` member function that returns the direction from this `Position` to an adjacent `Position`.

   For this question, Potential Movement Locations are positions that are:

   (1) adjacent to the current fish position,
   (2) in the direction the fish is moving, `myDir`, or 45 degrees to the right or left of `myDir`, and
   (3) empty and in the environment `env`.

   The diagram below shows three fish, represented by arrows showing their current directions. The Potential Movement Locations for each fish are shaded. For example, for the fish at position (2,1) in the diagram with `myDir` equal `"E"` (as indicated by the arrow), Potential Movement Locations would be those positions to the northeast, east, and southeast that are empty. For the fish at position (2,9) with `myDir` equal `"SW"`, Potential Movement Locations would be those positions to the west, southwest and south that are empty.

Potential Movement Locations

	0	1	2	3	4	5	6	7	8	9	10	11
0					NW	N	NE					
1	NW	N	NE		W	↗	E		NW	N	NE	
2	W	→	E		SW	S	SE		W	↙	E	
3	SW	S	SE						SW	S	SE	
4												

The `Fish` class is modified to include an additional private data member to hold the direction in which the fish is moving and a private member function to determine the positions of legal forward moves for the fish. The modifications to the `Fish` class are shown below.

```
class Fish
{
  public:
    // ... member functions as in the original version

  private:
    // ... private data as in the original version

    apstring myDir; // "N" for north, "E" for east,
                    // "S" for south, "W" for west,
                    // "NE" for northeast, "SE" for southeast,
                    // "NW" for northwest, "SW" for southwest

    Neighborhood ForwardNbrs(const Environment & env) const;
    // postcondition: returns empty neighbors in a forward direction
    //                from myDir - straight ahead and diagonally ahead
    //                to the right or left

};
```

Four new public member functions are added to the `Position` class: `Northeast()`, `Southeast()`, `Northwest()`, and `Southwest()`, each of which returns the neighboring `Position` in the specified direction. Functions `North`, `East`, `South`, and `West` are unchanged. The modifications to the `Position` class are shown below.

```
class Position
{
  public:
    // ... member functions as in the original version

    Position Northeast() const;
      // postcondition: returns Position northeast of this position
    Position Southeast() const;
      // postcondition: returns Position southeast of this position
    Position Northwest() const;
      // postcondition: returns Position northwest of this position
    Position Southwest() const;
      // postcondition: returns Position southwest of this position

    apstring DirectionTo(const Position & other) const;
    // precondition:  other is adjacent to this Position
    // postcondition: returns the string representation of the direction
    //                from this Position to other

  private:
    // ... private data as in the original version

};
```

(a) You will write the `Position` member function `Northeast`, which is described as follows. `Northeast` should return the position in the environment to the northeast of the current position. In the diagram shown above, if `pos1` is the position (2, 1), the call `pos1.Northeast()` returns the position (1, 2), and if `pos2` is the position (2, 9), the call `pos2.Northeast()` returns the position (1, 10).

Complete function `Northeast` below.

```
Position Position::Northeast() const
// postcondition: returns Position northeast of this position
```

(b) You will write the `Fish` member function `ForwardNbrs`, which is described as follows. `ForwardNbrs` should return a neighborhood consisting of those positions that meet the requirements for Potential Movement Locations.

In writing `ForwardNbrs`, you may use any of the `Fish` and `Position` member functions. Assume that these functions, including `Position::Northeast`, work as specified, regardless of what you wrote in part (a).

An implementation of this function distinguishes among multiple cases based on direction. In writing your code, you must show the code for the <u>two</u> specific cases, <u>north</u> and <u>northeast</u>. You may write `" ... "` to indicate where the remaining cases should be. All statements other than these remaining cases must be shown.

Complete function `ForwardNbrs` below.

```
Neighborhood Fish::ForwardNbrs(const Environment & env) const
// postcondition: returns empty neighbors in a forward direction from
//                myDir - straight ahead and diagonally ahead to the
//                right or left
```

(c) You will write the `Position` member function `DirectionTo`, which returns the direction from this `Position` to `Position other`.

An implementation of this function distinguishes among multiple cases based on direction. In writing your code, you must show the code for the <u>two</u> specific cases, <u>north</u> and <u>northeast</u>. You may write `" ... "` to indicate where the remaining cases should be. All statements other than these remaining cases must be shown.

Complete function `DirectionTo` below.

```
apstring Position::DirectionTo(const Position & other) const
// precondition:  other is adjacent to this Position
// postcondition: returns the string representation of the direction
//                from this Position to other
```

4. Consider the problem of assigning passengers to seats on airline flights. Three types of information are needed —passenger information, seat information, and flight information. Three classes will be used to represent this information, respectively: `Passenger`, `Seat`, and `Flight`.

   You will write three member functions for the `Flight` class:

   (a) `EmptySeatCount` that returns the number of empty seats of a specified type,

   (b) `FindBlock` that returns information about the location of an empty block of seats, and

   (c) `AssignGroup` that attempts to assign a group of passengers to adjacent seats.

   Passenger information is abstracted by a class and includes a name and other information. A default passenger, used to indicate "no passenger" in a seat, has the empty string `""` as its name. The declaration for class `Passenger` is as follows.

```
class Passenger
{
  public:
    Passenger();    // default passenger with name ""

    apstring GetName() const;
    // postcondition: returns passenger's name

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

   Seat information includes the passenger assigned to the seat and the type of the seat ("window", "aisle", "middle"). The `Seat` function `GetPassenger` returns the passenger assigned to the seat; if the seat is empty, `GetPassenger` returns a default passenger. The declaration for the class `Seat` is as follows.

```
class Seat
{
  public:
    Passenger GetPassenger() const;
    // postcondition: returns passenger in this seat

    apstring GetType() const;
    // postcondition: returns the type of this seat

    void SetPassenger(const Passenger & p);
    // postcondition: assigns p to this seat (i.e., GetPassenger() == p)

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

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
```

(b) You will write the `Flight` member function `FindBlock`, which is described as follows. `FindBlock` searches for a block of `seatsNeeded` adjacent empty seats in the specified row. If such a block of seats is found, `FindBlock` returns the column index of the first (i.e., the lowest index) seat in the block; otherwise, it returns -1.

The seating diagram for passengers of `Flight ap2002` is repeated here for your convenience.

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

The following table shows several examples of calling `FindBlock` for `Flight ap2002 as` shown.

Function Call	Value Returned
`ap2002.FindBlock(0, 1)`	`2` or `4`
`ap2002.FindBlock(0, 2)`	`-1`
`ap2002.FindBlock(1, 2)`	`2`

Complete function `FindBlock` below.

```
int Flight::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
```

(c) You will write the `Flight` member function `AssignGroup`, which is described as follows. The parameter to the `Flight` member function `AssignGroup` is an array of passengers, `group`. These passengers require a block of adjacent seats in a single row. `AssignGroup` searches for `group.length()` adjacent empty seats in some row. If such a block of seats is found, the passengers in `group` will be assigned to those seats, and `AssignGroup` returns `true`. Otherwise, no passengers are assigned to seats, and `AssignGroup` returns `false`.

For example, the seats in `Flight ap314` are as shown in the first diagram below. If the array `adults` contains three passengers, the call `ap314.AssignGroup(adults)` makes no changes to `ap314` and returns `false`, because there is no block of three adjacent empty seats in a single row. On the other hand, suppose the array `kids` contains passengers `"Sam"` and `"Alex"`. The call `ap314.AssignGroup(kids)` will assign `"Sam"` and `"Alex"` to the seats shown in the second diagram below and return `true`.

Contents of `mySeats` for `ap314` before any call to `AssignGroup`

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

Contents of `mySeats` for `ap314` after call to `ap314.AssignGroup(kids)`

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

In writing `AssignGroup`, you may call `FindBlock` specified in part (b). Assume that `FindBlock` works as specified, regardless of what you wrote in part (b).

Complete function `AssignGroup` below.

```
bool Flight::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
```

END OF EXAMINATION