Chapter 1 Data Abstraction: The Walls

1

```
const CENTS_PER_DOLLAR = 100;

/** Computes the change remaining from purchasing an item costing
    dollarCost dollars and centsCost cents with d dollars and c cents.
    Precondition: dollarCost, centsCost, d and c are all nonnegative
    integers and centsCost and c are both less than CENTS_PER_DOLLAR.
    Postcondition: d and c contain the computed remainder values in
    dollars and cents respectively. If input value d < dollarCost, the
    proper negative values for the amount owed in d dollars and/or c
    cents is returned. */
void computeChange(int dollarCost, int centsCost, int& d, int& c);
```

2a

```
const MONTHS_PER_YEAR = 12;
const DAYS_PER_MONTH[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

/** Increments the input Date values (month, day, year) by one day.
    Precondition: 1 <= month <= MONTHS_PER_YEAR,
                  1 <= day <= DAYS_PER_MONTH[month - 1], except
                  when month == 2, day == 29 and isLeapYear(year) is true.
    Postcondition: The valid numeric values for the succeeding month, day,
                   and year are returned. */
void incrementDate(int& month, int& day, int& year);

/** Determines if the input year is a leap year.
    Precondition: year > 0.
    Postcondition: Returns true if year is a leap year; false otherwise. */
bool isLeapYear(int year);
```

3a Change the purpose of an appointment:

```
changeAppointmentPurpose(apptDate: Date, apptTime: Time,
                         purpose: string): boolean
   if (isAppointment(apptDate, apptTime))
      cancelAppointment(apptDate, apptTime)

   return makeAppointment(apptDate, apptTime, purpose)
```

3b Display all the appointments for a given date:

```
displayAllAppointments(apptDate: Date)
   time = startOfDay

   while (time < endOfDay)
      if (isAppointment(apptDate, time))
         displayAppointment(apptDate, time)

      time = time + halfHour
```

This implementation requires the definition of a new operation, *displayAppointment*(), as well as definitions for the constants *startOfDay*, *endOfDay* and *halfHour*.

4

```
   Bag<string> fragileBag;

   while (storeBag.contains("eggs"))
   {
      storeBag.remove("eggs");
      fragileBag.add("eggs");
   }  // end while

   while (storeBag.contains("bread"))
   {
      storeBag.remove("bread");
      fragileBag.add("bread");
   }  // end while

   // Transfer remaining items from storeBag to groceryBag;
   Bag<string> groceryBag;
   v = storeBag.toVector();
   for (int i = 0; i < v.size(); i++)
      groceryBag.add(v.at(i));
```

5

```
/** Removes and counts all occurrences, if any, of a given string
 from a given bag of strings.
 @param bag  A given bag of strings.
 @param givenString  A string.
 @return  The number of occurrences of givenString that occurred
 and were removed from the given bag. */
int removeAndCount(ArrayBag<string>& bag, string givenString)
{
   int counter = 0;
   while (bag.contains(givenString))
   {
      counter++;
      bag.remove(givenString);
   }  // end while

   return counter;
}  // end removeAndCount
```

6

```
/** Creates a new bag that combines the contents of this bag and a
 second given bag without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the union of the two bags. */
public BagInterface<ItemType> union(BagInterface<ItemType> anotherBag);
```

7

```
/** Creates a new bag that contains those objects that occur in both this
 bag and a second given bag without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the intersection of the two bags. */
public BagInterface<ItemType> intersection(BagInterface<ItemType> anotherBag);
```

8

```
/** Creates a new bag of objects that would be left in this bag
 after removing those that also occur in a second given bag
 without affecting the original two bags.
 @param anotherBag  The given bag.
 @return  A bag that is the difference of the two bags. */
public BagInterface<T> difference(BagInterface<T> anotherBag);
```

9a

```
display(p.coefficient(p.degree()))
```

9b

```
p.changeCoefficient(p.coefficient(3) + 8, 3)
```

9c

```
for (power = 0; power < p.degree() || power < q.degree(); power++)
     // R is the sum of polynomials P and Q to degree power.
     r.changeCoefficient(p.coefficient(power) + q.coefficient(power), power)
```

Chapter 2 Recursion: The Mirrors

---

1 The problem is defined in terms of a smaller problem of the same type:
Here, the last value in the array is checked and then the remaining part of the array is passed to the function.

Each recursive call diminishes the size of the problem:
The recursive call to `getNumberEqual` subtracts 1 from the current value for `n`, passing this as the parameter `n` in the next call, effectively reducing the size of the unsearched remainder of the array by 1.

An instance of the problem serves as the base case:
Here, the case where the size of the array is 0 (i.e.: $n \leq 0$) results in the return of the value 0: an array of size 0 can have no instances of the `desiredValue`. This terminates the recursion.

As the problem size diminishes, the base case is reached:
`n` is an integer and is decremented by 1 with each recursive call. After `n` recursive calls, the parameter `n` in the *n*th call will have the value 0 and the base case will be reached.

2a The call `rabbit(5)` produces the following box trace: