Algorithms and Parallel Computing

POLITECNICO
MILANO 1863

Course 052496
Prof. Danilo Ardagna, Prof. Matteo Giovanni Rossi

Date: 16-01-2025

Last Name: .....
First Name: .....
Student ID: .....
Signature: .....

Exam duration: 2 hours and 30 minutes

Students can use a pen or a pencil for answering questions.

Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.

Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.

Writing on the cheat sheet is NOT allowed.

Exercise 1: _____ Exercise 2: _____ Exercise 3: _____

Exercise 1 (13 points)

You are required to implement a library system that manages the lending of various items across libraries located in different places. The system is centralized, enabling libraries to share items and allowing users to borrow and return items at any library in the system.

Libraries store three types of items: books, DVDs, and magazines. `Item` (graphically depicted in Figure 1) is an *abstract* class containing general information about an item, such as its unique ID, state, and in how many days it will become available. The state of an item is defined as follows:

```
1 enum State {  LOANED, AVAILABLE };
```

Note that an item is AVAILABLE when it is physically present at a library, whereas it is LOANED when it is held by a user. Class `Item` implements the following function:

- `return_item`: it updates the state and the `days_to_availability` of a LOANED item when it is returned to a library in the system. It returns `true` if the operation is successful (i.e., the LOANED item is returned), `false` otherwise.

The class also has the usual getter methods, which return the values of the private attributes.

Class `Item` is specialized into `Book`, `Dvd` and `Magazine` (see Figure 1). Each sub-class contains item-specific information (e.g., for a book, the title, authors, and year of publication) and provides a specific implementation of the `lend` function. In particular, the `lend` function implements a distinct policy for each item type. This policy varies by the maximum number of days an item can be lent, which is defined by the LOAN_DURATION constant expression.

A library is represented by the `Library` class (see Figure 2). Each `Library` is characterized by its `location_id`, representing the ZIP code of the city where the library is located. Additionally, class `Library` stores the IDs (and only the IDs) of the items that are physically present or have been loaned by the library, as well as the distances (in days) to the other libraries in the system. Class `Library` implements the following methods:

- `add_item` and `remove_item` to add/remove an `item_id` to/from the inventory.
- `find_item`, which returns `true` if an `Item` with the ID passed as argument is part of the library inventory, `false` otherwise.
- `get_distance`, which takes as input the ID of a library and returns the distance in days for delivering an item to that library from the current one.

Important: an item can be borrowed from a library other than the one in which it is physically stored. In this case the item is considered part of the inventory of the pickup library starting from the moment when the

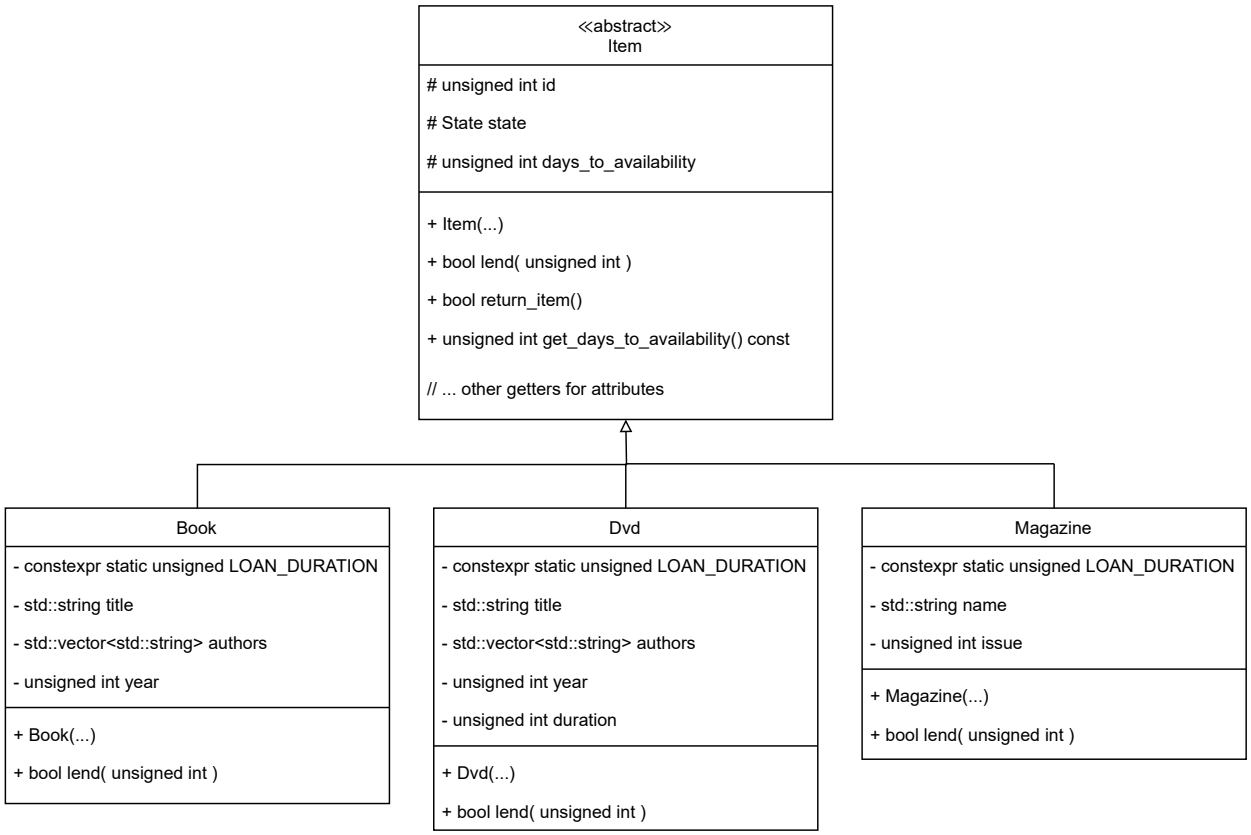

Figure 1: Class `Item` and its specializations.

borrowing request is made.

Important: only books and DVDs can be transferred between libraries. Magazines must be borrowed from and returned to the library where they are physically located.

The `Catalogue` class (also depicted in Figure 2) stores all the items managed by the system, categorized as books, DVDs, and magazines. It implements the following methods:

- `lend_item`, which takes the `item_id` and the `transit_days` (the days required for transit between libraries) as input, and returns a Boolean indicating whether the lending operation of the item is successful (i.e., the `AVAILABLE` item is loaned) or not (i.e., the item is already `LOANED`).
- `return_item`, which takes the `item_id` as input, and returns a Boolean indicating whether the return operation of the item is successful.
- `find_book`, `find_dvd` and `find_magazine`, which check, respectively, whether a book, a DVD, or a magazine with the given ID is present in the `Catalogue`.
- `get_books`, `get_dvds` and `get_magazines`, which return, respectively, a reference to attribute `books`, `dvds` and `magazines`.

Finally, the system is managed by class `Manager` (see Figure 2), which stores information regarding the libraries and the catalogue. It also provides a few methods, which are described below, and which you are asked to implement.

Considering the descriptions provided above, your task is to:

1. Provide the declaration of attributes `Library::items`, `Catalogue::books`, `Catalogue::dvds` and `Catalogue::magazines`, considering that items are searched by ID.
   Ensure that the types minimize the **worst-case** complexity of the search by ID.

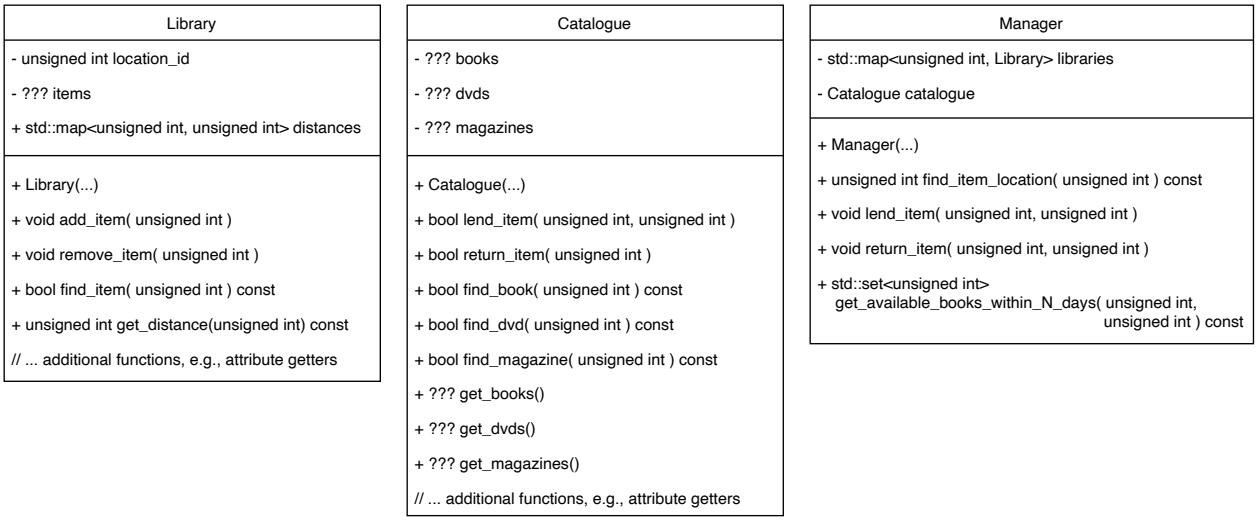

Figure 2: Classes `Library`, `Catalogue` and `Manager`.

2. Complete the declarations of the `lend(unsigned int transit_days)` functions within classes `Item`, `Book`, `Dvd` and `Magazine`.

Considering that **no class has the *default constructor***, provide the implementation of the following methods:

3. `bool Book::lend(unsigned int transit_days)`
   This function implements the lending policy specific for books. It updates attributes `state` and `days_to_availability` of the book (if it is available), considering the days of transit from one library to another (captured by parameter `transit_days`), which are added to the days of borrowing. The function returns `true` if the operation is successful (i.e., if the book is borrowed), `false` otherwise.

4. `bool Catalogue::find_dvd(unsigned int item_id) const`
   This function returns `true` if a DVD with the specified `item_id` exists in the catalogue, `false` otherwise.

5. `unsigned int Manager::find_item_location(unsigned int item_id) const`
   It returns the ID of the library where the item with ID `item_id` is stored or where it has been picked up by a user. If the item is not present in the system, the function returns 0 and prints an error on standard output.

6. `void Manager::lend_item(unsigned int item_id, unsigned int arrival_location)`
   It handles the borrowing of an item identified by `item_id`. Notice that the `arrival_location`, which is the ID of the library from which the user wants to pick up the item, may differ from the library where the item is currently stored (if available).
   The function prints an error on the standard output and does not do anything else if the item cannot be borrowed from the desired library.

7. `std::set<unsigned int> Manager::get_available_books_within_N_days( unsigned int N, unsigned int library_id) const`
   This function returns the IDs of the books available for loan and pick up within `N` days in the library identified by `library_id` (considering also that books can be transferred from one library to another).

Please note that for the implementation, you can rely on all the functions shown in Figure 1 and Figure 2.

8. Compute the **average-case complexity** of the methods implemented at points 4 and 5.

Solution 1

1. To minimize the worst-case complexity of the `find` operation, we rely on `std::set` and `std::map`. In particular, items in `Library` are stored only by ID, while in `Catalogue` you have to store the IDs and the items themselves. The declarations follow:
   `std::set<unsigned int> items`

std::map<unsigned int, Book> books
std::map<unsigned int, Dvd> dvds
std::map<unsigned int, Magazine> magazines

2. Item is an abstract class, and there is not a general policy for lending an Item; hence, lend must be a pure virtual method. Hence, the declaration of the lend function is the following:
   virtual bool lend(unsigned int transit_days) = 0
   Moreover, the method is overridden in the Book, Dvd and Magazine classes. For this reason, the complete declaration of method lend in those classes is the following:
   bool lend(unsigned int transit_days) override

3. bool Book::lend(unsigned int transit_days)
   This function must first check if the book is available for loan. If not, it is sufficient to return false. Otherwise, the function sets the days_to_availability to the sum of parameter transit_days and attribute LOAN_DURATION, it updates the state of the book to LOANED, and then returns true to indicate that the operation has been successful.

```
bool Book::lend(unsigned int transit_days) {
    if (state == State::AVAILABLE){
        days_to_availability = transit_days + LOAN_DURATION;
        state = State::LOANED;
        return true;
    }
    else {
        std::cerr << "Book already on loan" << std::endl;
        return false;
    }
}
```

4. bool Catalogue::find_dvd(unsigned int item_id) const
   This function is used to check whether a DVD identified by item_id exists in the catalogue. Since we defined Catalogue::dvds as std::map<unsigned int, Dvd>, it is sufficient to call the find function and check whether the iterator returned by find() is different from the one returned by end().

```
bool Catalogue::find_dvd(unsigned int item_id) const {
    return dvds.find(item_id) != dvds.end();
}
```

   Complexity: the map::find() function has average-case complexity $O(\log(D))$, where D is the number of DVDs in the Catalogue.

5. unsigned int Manager::find_item_location(unsigned int item_id) const
   To find the location of an Item we have to scan, sequentially, all libraries in the system. If the Item with ID item_id is stored (or has been picked up) in a specific library, then we return the ID of the latter, otherwise we return 0.

```
unsigned int Manager::find_item_location(unsigned int item_id) const {

    for (const auto &[id, library]: libraries){
        if (library.find_item(item_id))
            return id;
    }

    std::cerr << "Item not found";
    return 0;

}
```

   Complexity: let L be the number of libraries and N the number of items in a library. The average case complexity is $O(L\ log(N))$, as you have to loop through on average on half libraries and to call the set::find() function on each Library::items.

6. **void** Manager::lend_item(**unsigned int** item_id, **unsigned int** arrival_location)
   This function handles the borrowing of any **Item** within the **Catalogue**. It is important to consider that a **Magazine** cannot be transferred to a different **Library**, and that a user can borrow an **Item** from a **Library** different from the pick-up location. Recall that **find_item_location** returns 0 if it does not find the item in any library. Moreover, the function **Catalogue::lend_item** returns **true** if the item is lent successfully, otherwise it returns **false**, and the transfer of the item does not take place.

```
void Manager::lend_item(unsigned int item_id, unsigned int arrival_location) {

    // Retrieve departure/arrival locations
    unsigned int departure_location = find_item_location(item_id);

    if (departure_location == 0){
        std::cerr << "Item not found in the system" << std::endl;
        return;
    }

    // If the item is a magazine, you cannot send to another library
    if (catalogue.find_magazine(item_id) and departure_location != arrival_location){
        std::cerr << "You cannot transfer a magazine to a different library" << std::endl;
        return;
    }

    // Manage the loan
    if (catalogue.lend_item(item_id, libraries.at(departure_location).get_distance(arrival_location))
        and departure_location != arrival_location){
            libraries.at(departure_location).remove_item(item_id);
            libraries.at(arrival_location).add_item(item_id);
    }

}
```

7. std::set<**unsigned int**> Manager::get_available_books_within_N_days(**unsigned int** N,
   unsigned int library_id) **const**
   To implement this function you have to pay attention to the fact that a **Book** is available before N days if the sum of the **days_to_availability** and the possible **transfer_time** days is less than N.

```
std::set<unsigned int> Manager::get_available_books_within_N_days( unsigned int N,
                                                    unsigned int library_id ) const {

    std::set<unsigned int> books_;

    for (const auto& [item_id, book]: catalogue.get_books()){
        unsigned int days_to_availability = book.get_days_to_availability();
        unsigned int transfer_time = libraries.at(find_item_location(item_id)).get_distance(library_id)
     ;

        if (days_to_availability + transfer_time <= N)
            books_.insert(item_id);
    }

    return books_;

}
```

Exercise 2 (13 points)

We want to find the minimum and maximum in various "buckets" of non-negative integer numbers. More precisely, we consider that all integer numbers in the range $[0, 99]$ belong to one bucket, those in the range $[100, 199]$ in another, etc. (that is, each bucket is of the form $[100k, 100k + 99]$, for some $k \in \mathbb{N}$).

We want to write a program that performs the following operations:

(i) Reads from standard input a sequence of non-negative integer numbers in the range $[0, 499]$ (the sequence terminates when the user inputs a value outside of the specified range, or when he/she inputs something that is not a number).

(ii) For each of the 5 buckets $[100k, 100k + 99]$, with $k \in \{0, 1, 2, 3, 4\}$, it determines the minimum and maximum values that belong to the bucket, considering only the 2 *least significant digits* (e.g., if the maximum is 278, we only consider 78 as the value); if in some bucket there is no value, we conventionally consider 100 to be the minimum, and -1 to be the maximum.

We want to exploit a parallel architecture to realize the program.

1. Describe in your own words what strategy you want to use to paralellize the computation (i.e., how you distribute data across processes, what computation each process performs, how you generate the results).
2. Write suitable MPI code that realizes the program above.

   Note: you can assume that the length of the sequence of input values is a multiple of the number of processes.

Solution 2

1. To compute the desired result in a parallel fashion, we can proceed in the following way:

   (i) We read the input sequence from the process of rank 0.

   (ii) If we indicate by `size` the number of processes and by `n` the number of elements read , we split the sequence in `size` sub-sequences of length `n / size` each, and we send them to the process through a *scatter* operation.

   (iii) Through a simple linear pass on its portion of input, each process computes the minimum and maximum for each bucket (only for the portion of input that it received), and stores the values in local arrays `mins` and `maxs`, each of length 5.

   (iv) The global minima and maxima are computed through two *reduce* operations (one for the minimum, and one for the maximum), which collect the result on the process of rank 0.

   (v) Finally, the process of rank 0 outputs the minimum and maximum for each bucket (notice that this step is not required by the text of the exercises, it is performed for clarity).

2. A possible piece of code implementing the mechanisms above is the following:

```
1  #include <iostream>
2  #include <mpi.h>
3  #include <vector>
4
5  int main (int argc, char *argv[])
6  {
7     MPI_Init (&argc, &argv);
8
9     int rank, size;
10    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
11    MPI_Comm_size (MPI_COMM_WORLD, &size);
12
13    std::vector<int> seq_values, loc_values;
14
15    int local_n = 0;
16
17    if (rank == 0){
```

```
18          // step (i)
19          // Read from standard input, stop when the element read is not a number,
20          // or it is outside the range
21          int val;
22          while( (std::cin >> val) && val >= 0 && val < 499 )
23                seq_values.push_back(val);
24
25          // for completeness' sake, even though the text of the exercise says that we can
26          // assume that the length of the input sequence is a multiple of the number of processes,
27          // in case it is not we pad the sequence of input values by repeating
28          // the first element of the sequence (this does not change the result)
29          if ( seq_values.size() % size != 0 )
30             seq_values.resize(seq_values.size() + size - (seq_values.size() % size), seq_values[0]);
31
32          // step (ii), on the side of the process of rank 0
33          // tell the processes of rank other than 0 how many elements have been read
34          local_n = seq_values.size() / size;
35          MPI_Bcast ( &local_n, 1, MPI_INT, 0, MPI_COMM_WORLD );
36
37          // Send each process a piece of the input
38          loc_values.resize(local_n);
39          MPI_Scatter ( seq_values.data(), local_n, MPI_INT, loc_values.data(), local_n, MPI_INT,
40                0, MPI_COMM_WORLD );
41       } else {
42          // step (ii), on the side of the processes of rank other than 0
43          // each process other than the one of rank 0 receives the number of elements read
44          MPI_Bcast ( &local_n, 1, MPI_INT, 0, MPI_COMM_WORLD );
45
46          // each process of rank nonzero receives its piece of the input data
47          loc_values.resize(local_n);
48          MPI_Scatter ( nullptr, local_n, MPI_INT, loc_values.data(), local_n, MPI_INT,
49                0, MPI_COMM_WORLD );
50       }
51
52       // step (iii)
53       // each process computes the minimum and maximum for each bucket, for the piece of input data
54       // that it received
55
56       // initially the minimum of each bucket is set to 100, and the maximum is set to -1; that is,
57       // each minimum and maximum is set out of range, so that when they are compared against
58       // a value read from the input, the latter will be selected
59       std::vector<int> mins(5, 100), maxs(5, -1);
60
61       // compare each element in the piece of input sequence received with the current
62       // minimum and maximum for the corresponding bucket
63       for ( int raw_v : loc_values ){
64          int v = raw_v % 100;
65          int bucket = raw_v / 100;
66          if ( v < mins[bucket] ) mins[bucket] = v;
67          if ( v > maxs[bucket] ) maxs[bucket] = v;
68       }
69
70       // each process has computed the minima and maxima, the results now need to be put together
71
72       // since we use the MPI_IN_PLACE option, we separate the call for the root process (the one of
73       // rank 0), from those of the other processes
74       if (rank > 0){
75          // step (iv), on the side of the processes of rank other than 0
```

```
76          // for processes other than the root one, the recvbuf is irrelevant when the MPI_IN_PLACE
77          // option is used, so we pass nullptr
78          MPI_Reduce ( mins.data(), nullptr, 5, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD );
79          MPI_Reduce ( maxs.data(), nullptr, 5, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD );
80      } else {
81          // step (iv), on the side of the process of rank 0
82          // The root process uses the MPI_IN_PLACE option
83          MPI_Reduce ( MPI_IN_PLACE, mins.data(), 5, MPI_INT, MPI_MIN,
84                       0, MPI_COMM_WORLD );
85          MPI_Reduce ( MPI_IN_PLACE, maxs.data(), 5, MPI_INT, MPI_MAX,
86                       0, MPI_COMM_WORLD );
87
88          // step (v)
89          // The text of the exercise does not ask to produce the results in output
90          // Still, for completeness' sake, we write the result of the computation
91          // on the standard output, to check that the outcome is correct.
92          //
93          // The output is produced by the root process
94          for ( int i = 0; i < 5; i++){
95              std::cout << "Bucket [" << i*100 << ", " << i*100+99 << "]: min = ";
96              if ( mins[i] < 100 )
97                  std::cout << mins[i]+i*100;
98              else
99                  std::cout << "undef";
100             std::cout << ", max = ";
101             if ( maxs[i] >= 0 )
102                 std::cout << maxs[i]+i*100;
103             else
104                 std::cout << "undef";
105             std::cout << std::endl;
106         }
107     }
108
109     MPI_Finalize ();
110     return 0;
111 }
```

Through MPI_Init the MPI environment is initialized. Then, the process of rank 0 reads the sequence of integers from the standard input. If the sequence length is not divisible by the number of processes, it is padded with its first element to ensure an even distribution (note that this operation does not change the final results and it is not mandatory; it is proposed here to make the code more general and able to cope with any sequence length). The size of the local data (`local_n`) is shared across all processes through the MPI_Bcast primitive, while MPI_Scatter distributes chunks of the input sequence to all processes. In this way, each process calculates the minimum and maximum values for the five predefined buckets and values are stored in the corresponding vectors. The overall minima and maxima for each bucket are reconciled by the final MPI_Reduce operation. The MPI_IN_PLACE option allows the root process (the one with rank 0) to use the same buffer for sending and receiving during reduction.

Exercise 3 (4 points)

In this exercise, we work with an amusement park system, where multiple zones share a common stock of tickets. The core of the system manages the stock using functions such as *sellTicket()*, *addTickets()*, and *resetStock()* and shares the ownership of objects through the implementation of shared pointers.

The system is designed to:

- Share the ticket stock between different zones within the amusement park;
- Modify the stock of tickets through different operations (sell, add);
- Reset the stock to a set configuration.

The `main` function is a practical demonstration of the functionalities of the system.

After carefully reading the code, you have to answer the following question –> what numbers are shown at the following rows?

(a) Row 75;

(b) Row 76;

(c) Row 82;

(d) Row 88.

Notice that each answer requires you to type **just a single number!** Moreover, it is mandatory for you to develop a solution motivating the results you achieved on the sheets (that you need to upload if you are attending this exam online). Only providing the final answers is insufficient to obtain points in this section.

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4
5  typedef std::map<std::string, unsigned int> TicketStock;
6  typedef std::shared_ptr<TicketStock> SharedTicketStock;
7
8  class AmusementParkZone {
9  private:
10     std::string zoneName;
11     SharedTicketStock stock;
12
13 public:
14     AmusementParkZone(std::string name, const SharedTicketStock &tickets)
15         : zoneName(name), stock(tickets) {
16         initializeStock();
17     }
18
19     void sellTicket(const std::string &ticketType) {
20         if (stock->count(ticketType) && stock->at(ticketType) > 0)
21             stock->at(ticketType)--;
22     }
23
24     void addTickets(const std::string &ticketType, unsigned int quantity) {
25         if (stock->find(ticketType) == stock->end())
26             stock->emplace(ticketType, quantity);
27         else
28             stock->at(ticketType) += quantity;
29     }
30
31     unsigned int availableTickets(const std::string &ticketType) const {
32         return stock->count(ticketType) ? stock->at(ticketType) : 0;
33     }
```

```
34
35      void resetStock() {
36          stock = std::make_shared<TicketStock>();
37      }
38
39  private:
40      void initializeStock() {
41          for (auto &entry : *stock)
42              entry.second += 5;
43      }
44  };
45
46  void modifySharedStock(SharedTicketStock stock) {
47      if (stock->find("regular") != stock->end()) {
48          stock->at("regular") = 20;
49      }
50      if (stock->find("kids") != stock->end()) {
51          stock->at("kids") = 5;
52      }
53  }
54
55  int main() {
56      SharedTicketStock parkStock = std::make_shared<TicketStock>();
57      parkStock->emplace("regular", 0);
58      parkStock->emplace("kids", 0);
59      parkStock->emplace("seniors", 0);
60
61      AmusementParkZone zoneA("Zone A", parkStock);
62      AmusementParkZone zoneB = zoneA;
63
64      zoneA.sellTicket("regular");
65      zoneB.sellTicket("regular");
66      zoneB.sellTicket("kids");
67
68      zoneB.addTickets("kids", 3);
69      AmusementParkZone zoneC(zoneA);
70      zoneC.addTickets("seniors", 7);
71      zoneC.addTickets("kids", 2);
72
73      modifySharedStock(parkStock);
74
75      std::cout << "Available 'regular' tickets in Zone A: " << zoneA.availableTickets("regular") << std::endl;
76      std::cout << "Available 'kids' tickets in Zone B: " << zoneB.availableTickets("kids") << std::endl;
77
78      zoneA.resetStock();
79      zoneB.resetStock();
80      zoneB.addTickets("regular", 10);
81
82      std::cout << "Available 'regular' tickets in Zone A: " << zoneA.availableTickets("regular") << std::endl;
83
84      zoneA.sellTicket("seniors");
85      zoneB.sellTicket("seniors");
86      zoneC.sellTicket("seniors");
87
88      std::cout << "Available 'seniors' tickets in Zone C: " << zoneC.availableTickets("seniors") << std::endl;
89
90      return 0;
91  }
```

Solution 3

(a) **20** - "regular" tickets were set to 5 during the construction of zoneA. Then, modifySharedStock sets "regular" tickets in the shared stock to 20;

(b) **5** - "kids" tickets were sold and then incremented, but modifySharedStock overwrites their count in the shared stock to 5, which is visible to zoneB;

(c) **0** - after calling resetStock on zoneA, its stock is replaced with an empty stock;

(d) **11** - zoneC has the shared stock where "seniors" tickets were incremented to 5+7=12 earlier. The call to zoneC.sellTicket("seniors") reduces the count by 1.