Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna, Prof. Damian Andrew Tamburri

Date: 21-06-2024

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

<u>Exam duration: 2 hours and 30 minutes</u>

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: _____

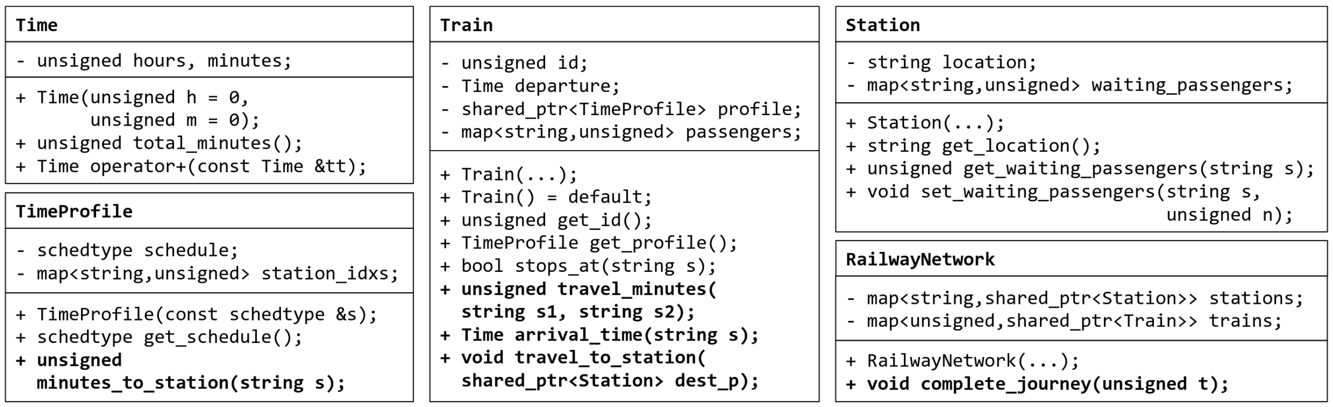
Exercise 1 (15 points)

You have to work on a program implementing a railway infrastructure for the simulation of passenger flows: its key elements shall be timetables, trains, and stations. In particular, the structure contains contains five main classes: `Time`, `TimeProfile`, `Train`, `Station`, `RailwayNetwork`. We describe these classes in the following, and represent their structure in the class diagram of Figure 1.

The `Time` class is used to represent time intervals and time differences (e.g., 0 hours and 8 minutes), as well as arrival and departure clock times (e.g., 18:25). Its constructor and **`operator+`** method perform the necessary conversions to ensure that these time representations are always appropriate, i.e., that the number of minutes is smaller than 60 (you can assume that `Time` objects never surpass the 24-hours threshold). Its `total_minutes` method returns the corresponding value converted to minutes; for example, for 1 hour and 8 minutes, it would return 68.

`TimeProfile` represents a timetable that a train follows when stopping at the stations along its path. The times in these timetables are represented as time differences with respect to the original departure time of the train, rather than with arrival/departure clock times. For instance, in Figure 2, the third row means that the train will be at Treviglio station 31 minutes after its original departure. This information is stored in the `schedule` member of type `schedtype`, which is declared with the following type definition:

```
typedef vector<pair<string, Time>> schedtype;
```



```
Time
- unsigned hours, minutes;
+ Time(unsigned h = 0,
       unsigned m = 0);
+ unsigned total_minutes();
+ Time operator+(const Time &tt);
```

```
TimeProfile
- schedtype schedule;
- map<string,unsigned> station_idxs;
+ TimeProfile(const schedtype &s);
+ schedtype get_schedule();
+ unsigned
  minutes_to_station(string s);
```

```
Train
- unsigned id;
- Time departure;
- shared_ptr<TimeProfile> profile;
- map<string,unsigned> passengers;
+ Train(...);
+ Train() = default;
+ unsigned get_id();
+ TimeProfile get_profile();
+ bool stops_at(string s);
+ unsigned travel_minutes(
  string s1, string s2);
+ Time arrival_time(string s);
+ void travel_to_station(
  shared_ptr<Station> dest_p);
```

```
Station
- string location;
- map<string,unsigned> waiting_passengers;
+ Station(...);
+ string get_location();
+ unsigned get_waiting_passengers(string s);
+ void set_waiting_passengers(string s,
                             unsigned n);
```

```
RailwayNetwork
- map<string,shared_ptr<Station>> stations;
- map<unsigned,shared_ptr<Train>> trains;
+ RailwayNetwork(...);
+ void complete_journey(unsigned t);
```

Figure 1: Class diagram. - is **`private`**, + is **`public`**. Methods to be implemented are highlighted in bold. **`const`** qualifiers for methods and the arguments of some constructors are omitted for the sake of brevity.

station name	Time
“Milano Lambrate”	0h08m
“Pioltello-Limito”	0h15m
“Treviglio”	0h31m
“Brescia”	1h08m

Figure 2: Example of `schedule` structure in a `TimeProfile` object.

In particular, the `schedule` contains station-`Time` pairs sorted in increasing order of `Time`, as shown in Figure 2. Note that times included in schedules are always non-null, i.e., strictly greater than zero minutes. The `TimeProfile` object also contains `station_idxs`, a `map` linking station names to their corresponding index number (as an **`unsigned`**) in the `schedule` vector; for instance, Treviglio station referenced in Figure 2 would have index 2. The `station_idxs` map is automatically filled by the constructor of the `TimeProfile` class. Note that a `map` that is separate from the `schedule` is needed in order to have efficient access by station name to schedule times.

The `Train` class represents an individual train that has a certain departure time (e.g., 18:25) and follows a given schedule (see, e.g., Figure 2). With respect to `Train`, schedules are in a separate class so that trains with different `departure` times (e.g., 18:25 and 19:25) can re-use the same `TimeProfile` object, which indeed is accessed via a `shared_ptr`. A `Train` object also contains a numeric identifier and `passengers`, which is a `map` reflecting information about the passengers currently on that train. In particular, `passengers[s]` is the number of passengers on that train that will get off the train at station `s`. Note that this `map` may be missing keys for some stations in the system; this obviously means that no passengers currently on the train will get off at that station. Finally, the `stops_at` method returns whether or not station `s` is included in the train timetable by looking up `s` in the `station_idxs` member of the profile.

A `Station` is identified by its `location` and also contains a `map` linking stations to passengers, `waiting_passengers` . Specifically, `waiting_passengers[s]` is the number of people waiting at the station who want to go to station `s`. Note that the `get_waiting_passengers` and `set_waiting_passengers` methods work correctly even if station `s` is not already present in the map (i.e., the former returns 0 while the latter creates a new `map` element).

Finally, the `RailwayNetwork` class holds `map` containers for all `Station` and `Train` objects (via `shared_ptr`s) in the system, linking these objects to their own identifiers.

For this exercise, **you may only use class methods that are explicitly mentioned in the class diagram; this includes getters and setters**. Other methods cannot be used unless they appear in the diagram. Moreover, you can assume that **every method listed on the diagram that is not requested for you to implement, is already implemented** and available for you to use.

Specifically, you have to implement the methods below. NOTE: their signatures are included in the class diagram, but **you have to add any missing `const` method qualifiers** to the signatures when writing your implementation.

1. `TimeProfile::minutes_to_station`, which returns the time (in minutes) needed to reach station `s` from the departure of the train, or 0 if the station is not present in the schedule;
2. `Train::travel_minutes`, which returns the time (in minutes) needed to get from station `s1` to station `s2`, or 0 if the request is not appropriate (i.e., a station is not present in the schedule or `s2` is reached before `s1`);
3. `Train::arrival_time`, which returns a `Time` object representing the arrival clock time to station `s`, or a `Time` object representing `0h0m` if the station is not present in the schedule;
4. `Train::travel_to_station`, which simulates the train moving to station `dest_p` (passed via a `shared_ptr`). This means updating the `passengers` member of `Train` and the `waiting_passengers` member of `Station`. In particular, all passengers directed to `dest_p` get off the train and leave the railway system, while all waiting passengers directed to stations that will be visited after `dest_p` (i.e., with larger schedule times than it) get on the train. You can assume that the train has unlimited passenger capacity. The method must **raise an error if `dest_p` is not present** in the schedule;
5. `RailwayNetwork::complete_journey`, which simulates the entire journey of train `t`, visiting all stations included in the schedule following the schedule order. It must **raise an error if train `t` does not exist**.

Finally, you have to:

6. compute the **average-case complexity** of **exactly three** implemented methods of your choice.

Solution 1

1. TimeProfile::minutes_to_station. Note that this is a **const** method. We find the iterator to the station using the appropriate map method. If the search is successful, we access the schedule index from the iterator, then the second element of the corresponding pair, then call total_minutes.

```
unsigned TimeProfile::minutes_to_station(string s) const {
  auto it = station_idxs.find(s);
  if (it == station_idxs.cend()) {
    return 0;
  }
  else {
    unsigned idx = it->second;
    return schedule[idx].second.total_minutes();
  }
}
```

   Complexity: let N be the number of rows in schedule. The only operation that is not time-constant is the single call to find, which in the map case has $O(\log(N))$ average complexity.

2. Train::travel_minutes. It is a **const** method that simply returns the difference in minutes_to_station return values, accessed via the profile pointer. We handle the error conditions with stops_at and a comparison between the two values.

```
unsigned Train::travel_minutes(string s1, string s2) const {
  if (!stops_at(s1) or !stops_at(s2))
    return 0;
  unsigned t1 = profile->minutes_to_station(s1);
  unsigned t2 = profile->minutes_to_station(s2);
  if (t1 > t2)
    return 0;
  else
    return t2 - t1;
}
```

   Complexity: since stops_at relies on map::find, as mentioned in the text, it has a $O(\log(N))$ complexity. The same holds for the two calls to minutes_to_station as seen above. This results in a total complexity of $O(\log(N))$.

3. Train::arrival_time. This is also a **const** method. Here, we can exploit the already implemented Time constructor, which has appropriate default values for a null Time object, and its **operator+**. Conversion to an appropriate clock time is handled automatically by these methods, as mentioned in the text.

```
Time Train::arrival_time(string s) const {
  if (!stops_at(s))
    return Time();
  return departure + Time(0, profile->minutes_to_station(s));
}
```

   Complexity: this method also has $O(\log(N))$ complexity due to calls at stops_at and minutes_to_station.

4. Train::travel_to_station. We first store the station name in a variable dest_loc for convenience, and immediately check whether the train reaches such station, returning immediately if this is not the case. The update of passengers[dest_loc] is straightforward, then we also save the schedule and distance from dest_loc for convenience. We loop on each station in the schedule, and we check whether the train still has to visit it by, again, comparing minutes_to_station return values. In this case, we get the number of waiting passengers for that station, set it to 0, and increase the passengers count for that station. Note that **operator+=** works even if the key does not appear in passengers, as the element is created on the fly with its default initialization value of 0.

```
void Train::travel_to_station(shared_ptr<Station> dest_p) {
  string dest_loc = dest_p->get_location();
  if (!stops_at(dest_loc)) {
```

```
4      std::cerr << "Error: train does not stop at " << dest_loc << std::endl;
5      return;
6    }
7
8    // Passengers directed to dest_loc get off the train
9    passengers[dest_loc] = 0;
10
11   // Get distance in minutes from dest_loc
12   schedtype schedule = profile->get_schedule();
13   unsigned dest_min = profile->minutes_to_station(dest_loc);
14
15   // Loop over schedule station-Time pairs
16   for (auto it = schedule.cbegin(); it != schedule.cend(); it++) {
17     string it_loc = it->first;
18     // If station comes after dest_loc, pick up passengers
19     if (profile->minutes_to_station(it_loc) > dest_min) {
20       unsigned wp = dest_p->get_waiting_passengers(it_loc);
21       // Update passengers in train
22       passengers[it_loc] += wp;
23       // Update waiting_passengers in station
24       dest_p->set_waiting_passengers(it_loc, 0);
25     }
26   }
27 }
```

Complexity: lines 2–13 have a total complexity of $O(\log(N))$. The loop scanning schedule sequentially executes N iterations, each of which is composed of a sum of several $O(\log(N))$ operations, since they all essentially reduce to lookups and insertions in maps. Therefore, the total complexity of the method is $O(N\log(N))$.

5. RailwayNetwork::complete_journey. After a check on the existence of t, we loop on the names of stations to visit by accessing the train schedule. We recover the corresponding pointer from the station map and call the travel_to_station member.

```
void RailwayNetwork::complete_journey(unsigned t) {
  if (trains.find(t) == trains.cend()) {
    std::cerr << "Error: train " << t << " does not exist" << std::endl;
    return;
  }
  // Get schedule which contains list of stations to visit
  shared_ptr<Train> tr = trains[t];
  schedtype schedule = tr->get_profile().get_schedule();
  // Loop over station-Time pairs
  for (auto s : schedule) {
    shared_ptr<Station> sp = stations[s.first];
    tr->travel_to_station(sp);
  }
}
```

Complexity: besides the initial $O(\log(N))$ calls to find and operator[ ], the complexity of the loop on the N elements of schedule is dominated by travel_to_station, which is $O(N\log(N))$ as explained in the previous point. We obtain an overall complexity of $O(N^2\log(N))$.

Exercise 2 (12 points)

Federated Learning is a field of Machine-Learning characterized by the presence of multiple agents (or parties) that cooperate to the learning process by sharing the outcomes of their individual updates.

As an example, for a linear regression problem whose goal is to learn the weights w_{ij} $\forall i, j = 1 \cdots n$ such that:

$$y = \sum_{i,j=1}^{n} w_{ij} x_{ij},$$

federated learning may be implemented by executing multiple iterations of the following algorithm:

1. Let each agent k learn a set of weights $\{w_{ij}^k\}_{i,j=1}^n$ based on the data $\{x_{ij}^k\}_{i,j=1}^n$ it has access to;

2. Share the acquired knowledge by applying a suitable *aggregation mechanism*, so that all agents end up with the same set of weights $\{w_{ij}^*\}_{i,j=1}^n$, which are used as starting point in the next iteration.

You have to implement two **parallel functions** that perform simple aggregation mechanisms reflecting the algorithm above as follows:

- The `fedavg` function, with the following prototype:

  `void fedavg(la::dense_matrix& weights);`

  the function is called by each agent passing as parameter the matrix of weights $\{w_{ij}^k\}_{i,j=1}^n$ resulting by the local training (step 1 of the algorithm above). The global weights $\{w_{ij}^*\}_{i,j=1}^n$ are then computed as the **average** of $\{w_{ij}^k\}_{i,j=1}^n$ over all the active agents.

- The `weighted_fedavg` function, with the following prototype:

  `void weighted_fedavg(la::dense_matrix& weights, double& accuracy);`

  the function receives, as second parameter, the `accuracy` reached by each agent during the local training. The global weights $\{w_{ij}^*\}_{i,j=1}^n$ are computed as the **average** of $\{w_{ij}^k\}_{i,j=1}^n$ over all the active agents, **weighted** by the corresponding accuracy. Moreover, an estimate of the **global accuracy** is also computed as the average of the local ones.

NOTE: **at the end of the aggregation process, all agents must obtain the updated global matrix of weights** $\{w_{ij}^*\}_{i,j=1}^n$ (and, possibly, the global accuracy).

The matrices of weights are represented using the `dense_matrix` class, whose declaration is reported below.

```
#ifndef DENSE_MATRIX_HH
#define DENSE_MATRIX_HH

#include <istream>
#include <vector>

namespace la // Linear Algebra
{
  class dense_matrix final
  {
    typedef std::vector<double> container_type;

  public:
    typedef container_type::value_type value_type;
    typedef container_type::size_type size_type;
    typedef container_type::pointer pointer;
    typedef container_type::const_pointer const_pointer;
    typedef container_type::reference reference;
    typedef container_type::const_reference const_reference;

  private:
    size_type m_rows = 0, m_columns = 0;
    container_type m_data;
    size_type
    sub2ind (size_type i, size_type j) const;

  public:
    dense_matrix (void) = default;

    dense_matrix (size_type rows, size_type columns,
                  const_reference value = 0.0);

    explicit dense_matrix (std::istream &);

    void
    read (std::istream &);

    void
    swap (dense_matrix &);

    reference
    operator () (size_type i, size_type j);
    const_reference
    operator () (size_type i, size_type j) const;

    size_type
    rows (void) const;
    size_type
    columns (void) const;

    dense_matrix
    transposed (void) const;

    pointer
    data (void);
    const_pointer
    data (void) const;

    void
    print (std::ostream& os) const;

    void
    to_csv (std::ostream& os) const;
  };

  dense_matrix
  operator * (dense_matrix const &, dense_matrix const &);

  void
  swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

The two functions can be implemented as follows:

- To compute the average of the weights matrices and make the result available to all agents (i.e., to all ranks), we can simply exploit the MPI_Allreduce routine. The local weights matrices are summed *in-place* to avoid creating useless copies, and then divided by the number of ranks (i.e., size) to get the average.

```
void fedavg(la::dense_matrix& weights)
{
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  unsigned nrows = weights.rows();
  unsigned ncols = weights.columns();

  MPI_Allreduce(
    MPI_IN_PLACE,
    weights.data(),
    nrows * ncols,
    MPI_DOUBLE,
    MPI_SUM,
    MPI_COMM_WORLD
  );

  for (unsigned i = 0; i < nrows; ++i)
    for (unsigned j = 0; j < ncols; ++j)
      weights(i,j) /= size;
}
```

- Finally, to compute the weighted average, each element of the `weights` matrix needs to be multiplied by `accuracy` before performing the `MPI_Allreduce` operation. Moreover, also the local accuracies need to be aggregated to serve as denominator in the average computation (and to compute the estimate of the global accuracy as required).

```
void weighted_fedavg(la::dense_matrix& weights, double& accuracy)
{
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  unsigned nrows = weights.rows();
  unsigned ncols = weights.columns();

  for (unsigned i = 0; i < nrows; ++i)
    for (unsigned j = 0; j < ncols; ++j)
      weights(i,j) *= accuracy;

  MPI_Allreduce(
    MPI_IN_PLACE,
    weights.data(),
    nrows * ncols,
    MPI_DOUBLE,
    MPI_SUM,
    MPI_COMM_WORLD
  );

  MPI_Allreduce(
    MPI_IN_PLACE,
    &accuracy,
    1,
    MPI_DOUBLE,
    MPI_SUM,
    MPI_COMM_WORLD
  );

  for (unsigned i = 0; i < nrows; ++i)
    for (unsigned j = 0; j < ncols; ++j)
```

```
      weights(i,j) /= accuracy;

  accuracy /= size;
}
```

Exercise 3 (3 points)

The following code models a taxi booking service where customers can book regular or luxury taxis for transportation. Each taxi has a unique ID and a brand associated with it. The system allows for tracking the availability of taxis and limits the number of passengers that can be accommodated in each taxi. The taxi booking service provides two types of taxis: `RegularTaxi` and `LuxuryTaxi`. Both types of taxis inherit from the base class `Taxi`.

- `RegularTaxi`: Represents a standard taxi service with a maximum capacity of 4 passengers.
- `LuxuryTaxi`: Represents a luxury taxi service with a maximum capacity of 2 passengers.

The `Taxi` class includes the following functions:

- `book()`: Allows customers to book a taxi.
- `printTaxiId()`: Prints the unique ID of the taxi.
- `isAvailable()`: Checks if the taxi is available for booking.
- `printBrand()`: Prints the brand of the taxi.

Each taxi maintains its availability status, allowing customers to know if a particular taxi is available for booking. When a customer requests to book a taxi, the system checks the availability of the taxi by checking if the number of passengers currently in the taxi is less than the maximum allowed passengers. If the taxi has not reached its passenger limit, the booking is confirmed. Otherwise, the booking is declined.

To update the number of passengers in the taxi, the function `updateNumPassengers` is called whenever a booking is confirmed and whenever a passenger leaves the taxi. That function receives a boolean specifying the feasibility of the operation and another parameter specifying the number of passengers to add or let out. A `LuxuryTaxi` can only be taken by two people at the same time, and they also must leave the taxi at the same time. For the `RegularTaxi`, there is no such constraint. The complete implementation of all the classes is reported below.

After carefully reading the code, you have to answer the following questions. **Notice that each answer requires you to write a single number or printed output!** Please make sure to mark these answers clearly on the sheets. Moreover, it is mandatory for you to **develop your solution motivating the results you achieved on the sheets. Only providing the final answers is not enough to obtain points** in this section.

1. What is the value printed at line 11 when calling `printTaxiId()` on `regularTaxi1`?
2. What is printed on the standard output at line 22 after calling the function `regularTaxi1->book()`?
3. What is printed at line 32?

- **Taxi.hpp**

```
#ifndef QUESTION3_TAXI_HPP
#define QUESTION3_TAXI_HPP

#include <string>

class Taxi {
protected:
    int taxiId;
    std::string brand;
    int numPassengers;

public:
    Taxi(int id, const std::string& brand);
    Taxi(const Taxi& other);
    virtual bool book() = 0;
    void printTaxiId();
    virtual bool isAvailable() = 0;
    virtual void printBrand();
    void updateNumPassengers(bool statusBooking, int num);
    int getNumPassengers() const;
};

#endif //QUESTION3_TAXI_HPP
```

- **Taxi.cpp**

```
#include "taxi.hpp"
#include <iostream>

Taxi::Taxi(int id, const std::string& brand) : taxiId(id), brand(brand),numPassengers(0) {}
Taxi::Taxi(const Taxi& other) : taxiId(other.taxiId), brand(other.brand), numPassengers(other.
     numPassengers) {
}

void Taxi::printTaxiId() {
    std::cout << "0" << std::endl;
}

void Taxi::printBrand() {
    std::cout << "None" << std::endl;
}

void Taxi::updateNumPassengers(bool statusBooking, int num){
    return;
}

int Taxi::getNumPassengers() const {
    return numPassengers;
}
```

- **RegularTaxi.hpp**

```
#ifndef QUESTION3_REGULARTAXI_HPP
#define QUESTION3_REGULARTAXI_HPP

#include "taxi.hpp"

class RegularTaxi : public Taxi {

private:
    static const int MAX_PASSENGERS = 4;
public:
    RegularTaxi(int id, const std::string& brand) : Taxi(id, brand) {}
    virtual bool book() override;
    virtual void printBrand() override;
    void printTaxiId() const;
    virtual bool isAvailable() override;
    void updateNumPassengers(bool statusBooking, int num);
};

#endif //QUESTION3_REGULARTAXI_HPP
```

- **RegularTaxi.cpp**

```
#include "regularTaxi.hpp"
#include <iostream>

bool RegularTaxi::book() {
    if (isAvailable()) {
        std::cout << "Regular taxi "<<taxiId<<" booked." << std::endl;
        return true;
    } else {
        std::cout << "Regular taxi "<<taxiId<<" is not available." << std::endl;
        return false;
    }
}

void RegularTaxi::printBrand() {
    std::cout <<"Regular taxi:"<<brand<< std::endl;
}
void RegularTaxi::printTaxiId() const {
    std::cout << taxiId << std::endl;
}

bool RegularTaxi::isAvailable() {
    return numPassengers < MAX_PASSENGERS;
}

void RegularTaxi::updateNumPassengers(bool statusBooking, int num){
    numPassengers += num;
}
```

- **LuxuryTaxi.hpp**

```
#ifndef QUESTION3_LUXURYTAXI_HPP
#define QUESTION3_LUXURYTAXI_HPP

#include "taxi.hpp"

class LuxuryTaxi : public Taxi {
private:
    static const int MAX_PASSENGERS = 2;
public:
    LuxuryTaxi(int id, const std::string& brand) : Taxi(id, brand) {}
    virtual bool book() override;
    virtual void printBrand() override;
    virtual bool isAvailable() override;
    void updateNumPassengers(bool statusBooking, int num);
};

#endif //QUESTION3_LUXURYTAXI_HPP
```

- **LuxuryTaxi.cpp**

```
#include "LuxuryTaxi.hpp"
#include <iostream>

bool LuxuryTaxi::book() {
    if (isAvailable()) {
        std::cout << "Luxury taxi "<<brand<<":"<<taxiId<<" booked." << std::endl;
        return true;
    } else {
        std::cout << "Luxury taxi "<<brand<<":"<<taxiId<<" is not available." << std::endl;
        return false;
    }
}

void LuxuryTaxi::printBrand() {
    std::cout <<"Luxury taxi:"<<brand<< std::endl;
}
bool LuxuryTaxi::isAvailable() {
    return numPassengers < MAX_PASSENGERS;
}
void LuxuryTaxi::updateNumPassengers(bool statusBooking, int num){
    if (statusBooking) {
        if (num > 0) {
            numPassengers += 2;
        } else {
            numPassengers -= 2;
        }
    }
}
```

- **main.cpp**

```
#include "regularTaxi.hpp"
#include "LuxuryTaxi.hpp"
#include <iostream>

int main() {

    Taxi* regularTaxi1 = new RegularTaxi(101, "Toyota");
    RegularTaxi* regularTaxi2 = new RegularTaxi(102, "BMW");
    LuxuryTaxi* luxuryTaxi = new LuxuryTaxi(103, "Ford");

    regularTaxi1->printTaxiId();
    regularTaxi2->printTaxiId();
    luxuryTaxi->printTaxiId();
    luxuryTaxi->printBrand();

    bool statusBooking = regularTaxi1->book();
    regularTaxi1->updateNumPassengers(statusBooking,2);

    statusBooking = regularTaxi1->book();
    regularTaxi1->updateNumPassengers(statusBooking,2);

    statusBooking = regularTaxi1->book();
    regularTaxi1->updateNumPassengers(statusBooking,4);

    bool statusBooking2 = luxuryTaxi->book();
    luxuryTaxi->updateNumPassengers(statusBooking2,1);

    Taxi* taxi2 = luxuryTaxi;
    taxi2->updateNumPassengers(true,-1);
    std::cout<<taxi2->getNumPassengers()<<std::endl;

    regularTaxi1->printBrand();

    return 0;
}
```

Solution 3

1. **0**.
   The function `printTaxiId()` is defined in `Taxi`, `RegularTaxi` and `LuxuryTaxi`. But since it is not a defined with **`virtual`** keyword in `Taxi` there is no dynamic binding and the function `printTaxiId()` in `Taxi` is the one used.

2. **Regular taxi 101 booked**.
   In fact, one may expect the variable `numPassengers` to be 4 at line 22 and not being able to book the Taxi, but the `updateNumPassengers` is not declared as **`virtual`**, so at line 17 and 20, there where no update of `numPassengers` since the function that was called in `Taxi` does nothing.

3. **Regular taxi:Toyota**.
   `printBrand()` is a virtual function and it is overriden in both `RegularTaxi` and `LuxuryTaxi`. So with dynamic binding the function in `RegularTaxi` is the one called at line 32.