Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna
Prof. Damian Andrew Tamburri

Date: 13-06-2023

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 (12 points)

You work as the software developer of a molecular simulation program. In this program, a set of point-like particles with negligible mass move freely within the boundaries of a two-dimensional, continuous rectangular space, which will be referred to as "box" from now on. The four boundaries of the box lie at coordinates $x_{min}, x_{max}, y_{min}, y_{max}$ of the 2D Cartesian plane. Moreover, particles are uniquely identified by a progressive identifier (ID) number, and are characterized by (1) a coordinate pair in the Cartesian plane that determines their position, and (2) a velocity vector, as represented in Figure 1.

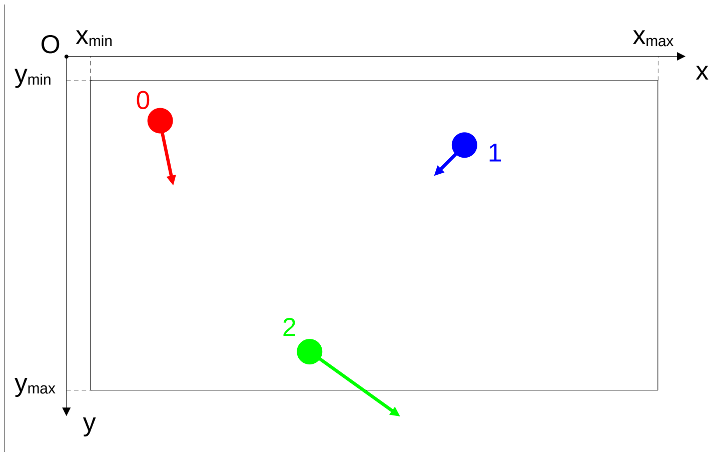


Figure 1: Representation of a molecular simulation with three particles.

When a particle hits one of the box walls, it experiences elastic collision, meaning that it simply bounces off the wall at the same angle, similarly to how reflections of electromagnetic waves work, or to what happens to billiard balls. Because of the "conservation of kinetic energy" principle, for each particle, the norm of its velocity vector—namely, its speed—stays the same at all times, even after collisions.

On the one hand, as a result of the collision—and considering that a vector is decomposable in its two basic components over the reference plane—one of the two components of the velocity vector (depending on the direction of the collision) shall change its sign, while the other will remain unchanged. On the other hand, collisions among particles cannot occur, since, as previously mentioned, they are assumed to be point-like.
The evolution of the system is handled via a time discretization. Specifically, if the total simulation time is $n\Delta t$ seconds, the code will sequentially perform n iteration steps, with each lasting Δt seconds. Figure 2 displays the

state of the system both at the initial time instant ($t = 0$ seconds) and after one second has passed ($t = 1$ second). During this time lapse, particle 2 experiences collision with the bottom wall of the box and bounces off of it, while particles 0 and 1 proceed normally in a straight direction.

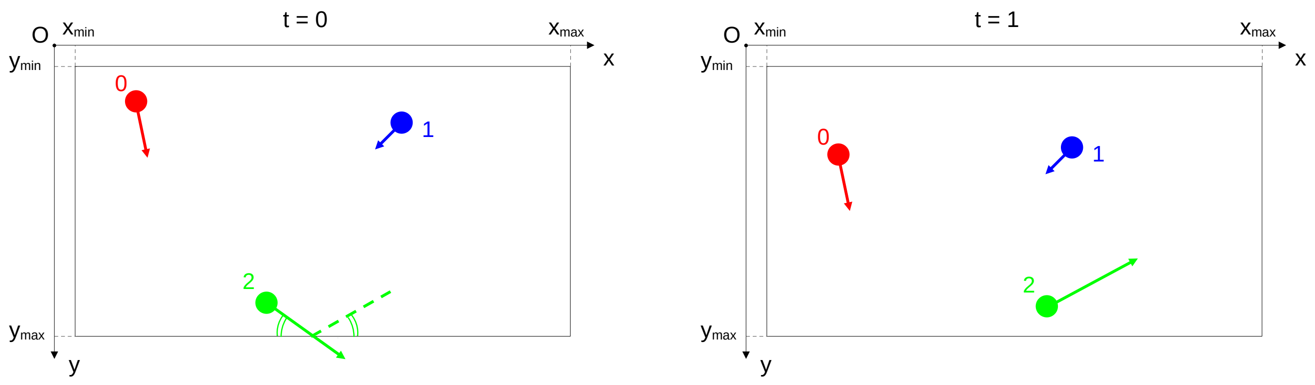
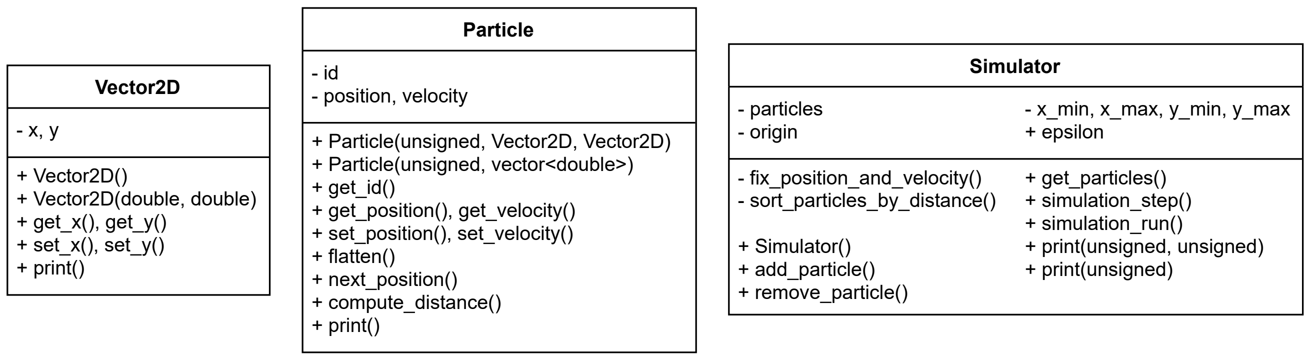


Figure 2: Example of system evolution. Here $n = 2$ and $\Delta t = 1$ second.

The code structure of the molecular simulation program is summarized in the class diagram of Figure 3.



Vector2D

- x, y

+ Vector2D()
+ Vector2D(double, double)
+ get_x(), get_y()
+ set_x(), set_y()
+ print()

Particle

- id
- position, velocity

+ Particle(unsigned, Vector2D, Vector2D)
+ Particle(unsigned, vector<double>)
+ get_id()
+ get_position(), get_velocity()
+ set_position(), set_velocity()
+ flatten()
+ next_position()
+ compute_distance()
+ print()

Simulator

- particles
- origin
- x_min, x_max, y_min, y_max
+ epsilon

- fix_position_and_velocity()
- sort_particles_by_distance()

+ Simulator()
+ add_particle()
+ remove_particle()
+ get_particles()
+ simulation_step()
+ simulation_run()
+ print(unsigned, unsigned)
+ print(unsigned)

Figure 3: Class diagram of the program (-: `private`, +: `public`). Note that method arguments have been omitted except for the overloaded constructors, in order to distinguish them.

In particular, the `Vector2D` class generalizes the notion of a coordinate pair of `double`s, and is used to represent both space positions and velocity vectors components. Each `Particle` object stores its own `position` and `velocity` vectors, as well as its `unsigned` ID number. Finally, the `Simulator` class holds a collection of `Particle` objects in the `particles` member.
At each simulation step, collisions are handled by first allowing the particles to move in the entire Cartesian plane, possibly even outside the box; then, if any particle does end up out of bounds, its position and velocity are updated to the correct values, taking the collision into account.
All methods appearing in the class diagram are **already implemented and available for you to use**, except the methods whose implementation is requested to you below. In particular, some of the already available members and methods are:

- `void Particle::next_position(double Dt)` which updates the particle position after it moves **freely** in the Cartesian plane for `Dt` seconds (which is the Δt described earlier). We stress that a particle by itself has no notion of a boundary, therefore this method **does not take any collisions into account**;
- `double Particle::compute_distance(const Vector2D& other_pos) const` which returns the Euclidean distance between the position of the current particle and `other_pos`;
- `const Vector2D Simulator::origin` which is a member representing the origin O of the Cartesian plane.

Your task is to work on the `Simulator` class. In particular, you have to:

1. provide the definition of the `particles` member with the aim to **optimize the average-case** complexity of the search by ID of a particle.

Moreover, you have to **implement the following methods**:

2. `void add_particle(const Particle& p)` which adds the given particle to the simulator. You first have to check whether the particle is within the boundaries of the box, up to a tolerance of `Simulator::epsilon` in all four directions. If out of bounds, you have to **print an error message** instead;
3. `void remove_particle(unsigned id)` which removes the particle with the given ID, if any, from the simulator;
4. `void fix_position_and_velocity(Particle& rp)` which receives a reference to a `Particle`. If the particle is out of bounds (meaning that a collision should have occurred), it updates its `position` and `velocity` so as to take the effects of the collision into account. The complete function takes care of any collision in all directions. However, for the purposes of this question, for the sake of simplicity, **only consider and handle collisions with the bottom wall** when writing the implementation (i.e., the case shown in Figure 2);
5. `void simulation_step (double Dt)` which performs a single step of the system evolution, advancing the state by `Dt` seconds and updating positions and velocities for all particles in the simulator;
6. ... `sort_particles_by_distance() const` which returns both the particles IDs sorted by increasing distance from origin, and their distances, **optimizing the operations worst-case complexity**. You are free to **choose any return type you like** for this function, provided that it fits the above requirements.

Finally, you have to:

7. compute the **worst-case complexity** of the `sort_particles_by_distance` method you have implemented.

Note that for all questions of the exercise, **you are free to implement new helper functions as members of the given classes**, if you need them.

Solution 1

1. The most appropriate data structure for the requested task is an `unordered_map`. We can therefore write in `Simulator.h`:

```
public:
  typedef std::unordered_map<unsigned, Particle> particle_container;

private:
  particle_container particles;
```

Some possible implementations of the required methods are reported below.

2. For `add_particle`, we can write a helper function `is_in_limits`:

```
bool Simulator::is_in_limits (const Particle& p) const
{
  const Vector2D & position = p.get_position();
  return x_min - epsilon <= position.get_x() &&
         position.get_x() <= x_max + epsilon &&
         y_min - epsilon <= position.get_y() &&
         position.get_y() <= y_max + epsilon;
}

void Simulator::add_particle (const Particle& p)
{
  if (!is_in_limits(p)){
    std::cerr << "particle out of bounds" << std::endl;
    return;
  }
  particles.insert({p.get_id(), p});
}
```

3. For `remove_particle`, we simply have:

```
void Simulator::remove_particle (unsigned id)
{
  Simulator::particle_container::iterator it = particles.find(id);

  if (it != particles.end())
      particles.erase(it);
}
```

4. For fix_position_and_velocity, if the particle is out of bounds with respect to the bottom wall y_max by a certain amount delta_pos, we shift its position inside the wall by the same amount. We also change the sign of the y component of its velocity vector due to the collision.
   The lines shown below are the only ones requested to you. As mentioned in the text, the complete function also handles the three other types of collision: with the top wall y_{min}, and with the left and right walls x_{min} and x_{max}. The code for these other cases is similar to the lines shown. (Note that after the y_{min} case, the function needs to update the position rpos before handling the horizontal collisions, as its y-component may have changed due to the vertical collisions.)

```
void Simulator::fix_position_and_velocity (Particle& rp)
{
  Vector2D rpos = rp.get_position();

  // Particle moving from up to down (recall: (x_min,y_min) is the top left corner)
  if(rpos.get_y() > y_max)
  {
    // update y coordinate
    double delta_pos = rpos.get_y() - y_max;
    Vector2D new_position(rpos.get_x(), y_max - delta_pos);
    rp.set_position(new_position);
    // update y velocity component
    Vector2D new_velocity (rp.get_velocity());
    new_velocity.set_y(-new_velocity.get_y());
    rp.set_velocity(new_velocity);
  }
```

5. In simulation_step, we simply loop on all particles in the simulator and call the two appropriate methods for the update:

```
void Simulator::simulation_step (double Dt)
{
  for (auto it = particles.begin(); it != particles.end(); ++it)
  {
    it->second.next_position(Dt);
    fix_position_and_velocity(it->second);
  }
}
```

6. For sort_particles_by_distance, an std::map which maps distances to particle IDs is the perfect candidate for the required task. We can simply create and fill the map, and the particles will automatically be sorted in the process. The text mentions that no ties occur in terms of distance, therefore we can be confident that the mapping is one-to-one. The following implementation exploits the already existing Particle::compute_distance method:

```
std::map<double, unsigned> Simulator::sort_particles_by_distance() const
{
  std::map<double, unsigned> new_distances;

  for (auto cit = particles.cbegin(); cit != particles.cend(); ++cit)
  {
    const auto & rp = cit->second;
    new_distances.insert({rp.compute_distance(origin), rp.get_id()});
    }

    return new_distances;
}
```

As for the worst-case complexity of `sort_particles_by_distance`:

7. Let N be the number of particles in the simulator. Insertion in a `map` has a $O(\log(N))$ complexity, while all other operations, including computation of the distance between the particle and `origin`, have constant complexity. Since the operations are performed for each particle, we have an overall complexity of $O(N \log(N))$.

Exercise 2 (14 points)

Your task is to work on all the necessary functions to parallelize the molecular simulation program of the previous exercise. A parallel simulation step consists of four sub-steps (also reported in Figure 4) which are recapped as follows:

0. A `vector` of `Particle`s within the box $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ is created.
1. A **local** `Simulator` is initialized by each process rank; such initialisation divides the box along the y-axis in a suitable number of *stripes*, and the `Particle`s are assigned to the corresponding local simulator depending on their position in the box and within a specific stripe.
2. Each local `Simulator` performs a simulation step, computing the new positions and velocities of all its corresponding `Particle`s. Note that, at the end of each simulation step, every `Particle` may be faced with one of two choices:

   - remain in the area managed by the current rank, or
   - end up in a different stripe, managed by a different rank.

   As an example, only the transitions observed by `rank 1` are reported in Figure 4: particle `p3` remains in its original rank stripe; particle `p4` exits the stripe and goes into another; particle `p1`—originally owned by `rank 2`—is transferred to the stripe of `rank 1`. The particles migration towards other stripes are managed in the next step.
3. Each rank:

   (a) determines which `Particle`s have left its local stripe after the simulation step, and

   (b) sends them to the relevant processes, receiving in turn the ones that entered its stripe as a consequence of the simulation step that just passed.

You have to work on the communication parts of this parallel step. In particular, **you have to:**

- implement the function:

  ```
  void send_particle(const Particle& particle, int dest);
  ```

  that receives as parameter a `Particle` and sends it to the given destination rank (`dest`).
- implement the function:

  ```
  Particle recv_particle(int src);
  ```

  that receives and returns the `Particle` communicated by the source rank (`src`) provided as parameter.

  For these first two points, remember that **a `Particle` cannot be directly communicated in MPI**. However, the `Particle` class provides a method:

  ```
  std::vector<double> Particle::flatten (void) const;
  ```

  which **returns a vector containing the four numbers representing the `Particle` position** (first two numbers) **and velocity** (last two numbers). Moreover, the class provides a **constructor that initializes a `Particle` given its `id` and the `vector` of its flattened representation**.

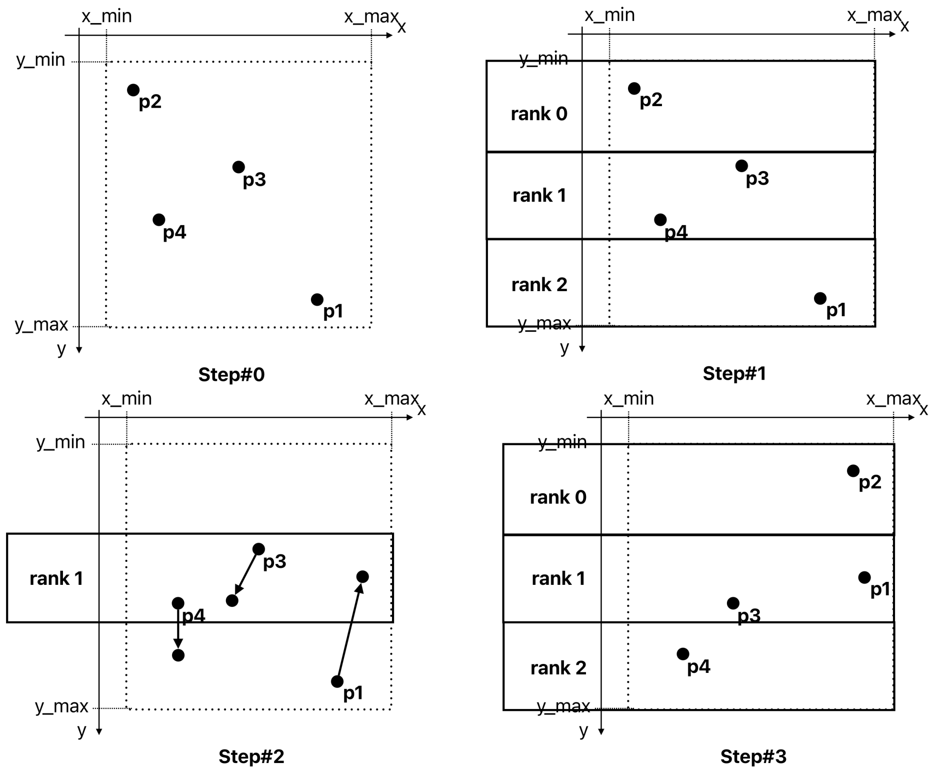


Figure 4: Parallel simulation step.

- complete the implementation of a **parallel function** with the following prototype:

```
void communicate_particles(int r, std::vector<std::vector<Particle>>& particles)
```

This function implements sub-step 3(b). In fact, it receives as parameters (1) the index `r` of the process that is currently sending `Particle`s to the other ranks, and (2) a container used to store the `Particle`s involved in the communication. Specifically, each `rank` stores in `particles[i]`, with `i != rank`, the `vector` of all `Particles` that it should send to process `i`, while it will save in `particles[rank]` the `vector` of `Particles` it receives. **The initial function implementation is provided at the end of the exercise, for you to complete.**

Considering the example in Figure 4, at the beginning of step 3(b) when `communicate_particles(1, particles)` is called (i.e., `rank 1` is the sender), the content of `particles` for `rank 1` is:

$$\begin{bmatrix} [\quad] \\ [\quad] \\ [p4] \end{bmatrix} \tag{1}$$

As shown in the figure, particle `p4` leaves the stripe managed by `rank 1` in Step#2 and needs to be sent to `rank 2`. On the other hand, particle `p3` remains in the same stripe and thus is not involved in the communication. Note that, for simplicity, `p4` **is not erased** from `particles` after the communication.

When, instead, `communicate_particles(2, particles)` is called (i.e., `rank 2` is the sender), the `particles` matrix for `rank 1` when the function execution terminates is:

$$\begin{bmatrix} [\quad] \\ [p1] \\ [p4] \end{bmatrix} \tag{2}$$

Indeed, `rank 1` receives particle `p1`, which entered its stripe as a consequence of the simulation.

In this example, the `particles[0]` vector is always empty because `rank 1` does not send to or receive particles from `rank 0`.

```
void communicate_particles(int r, std::vector<std::vector<Particle>>& particles)
{
    // YOUR CODE GOES HERE

    if (rank == r)
    {
      // rank r sends to each other_rank the particles stored in particles[other_rank]

      // YOUR CODE GOES HERE
    }
    else
    {
      // if the current rank is not r, it receives the particles from r and stores them in particles[rank]

      // YOUR CODE GOES HERE
    }
  }
```

Solution 2

The implementation of the three functions is reported in the following. Note that tags 0 through 2 are used in the proposed solution to distinguish the different parts of the communication process, but using a single tag is not an error.

-
  ```
  void send_particle(const Particle& particle, int dest)
  {
    // get the id and flatten the particle
    unsigned id = particle.get_id();
    flat_particle_t fp = particle.flatten();

    // send the id (tag: 1) and the flattened particle (tag: 2)
    MPI_Send(&id, 1, MPI_UNSIGNED, dest, 1, MPI_COMM_WORLD);
    MPI_Send(fp.data(), 4, MPI_DOUBLE, dest, 2, MPI_COMM_WORLD);
  }
  ```

  Note that, since a Particle cannot be communicated directly in MPI, the corresponding id and its flattened representation are sent separately.

-
  ```
  Particle recv_particle(int src)
  {
    // receive the particle id (tag: 1)
    unsigned id;
    MPI_Recv(&id, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // receive the flat particle (tag: 2)
    flat_particle_t fp(4);
    MPI_Recv(fp.data(), 4, MPI_DOUBLE, src, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
      ;

    return Particle(id, fp);
  }
  ```

  In the function, we receive the particle id and flattened representation, and we return a Particle object created via a suitable constructor.

-
  ```
  void communicate_particles(
    int r,
    std::vector<std::vector<Particle>>& particles
  )
  {
  ```

```
6    int rank, size;
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10   if (rank == r)
11   {
12     // rank r sends its particles to the others
13     for (int other_rank = 0; other_rank < size; ++other_rank)
14     {
15       if (other_rank != rank)
16       {
17         // send number of particles (tag: 0)
18         unsigned n_particles = particles[other_rank].size();
19         MPI_Send(&n_particles,1,MPI_UNSIGNED,other_rank,0,MPI_COMM_WORLD);
20
21         // loop over all particles
22         for (unsigned i = 0; i < n_particles; ++i)
23           send_particle(particles[other_rank][i], other_rank);
24       }
25     }
26   }
27   else
28   {
29     // all the other ranks receive from r
30
31     // number of particles (tag: 0)
32     unsigned n_particles;
33     MPI_Recv(&n_particles, 1, MPI_UNSIGNED, r, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
34
35     // loop over the number of particles to receive
36     for (unsigned i = 0; i < n_particles; ++i)
37     {
38       // receive the particle
39       Particle particle = recv_particle(r);
40
41       // add the received particle to the table
42       particles[rank].push_back(particle);
43     }
44   }
45 }
```

The function is divided in two main blocks: in the first one (lines 10–26), we check if the current rank is the one that should send particles to the others (i.e., `rank == r`). If so, we loop over all the other ranks and, for each one of them, we communicate:

- the number of particles `other_rank` will receive;
- in a loop, the current particle `particles[other_rank][i]`.

The second block (lines 27–45) is only entered if the current rank is not the sender, but the receiver (i.e., `rank != r`). In it, we receive all the particles sent by rank `r`, which are added to `particles[rank]`.

Exercise 3 (5 points)

The code below provides the definition for the enum `Item` that lists the available board games in stock at a board games shop. The class `Shop`, which contains a unique identifier for the shop, contains the set of available games in `list`, and `availableCopies` stores the number of games copies that are available for sale (which may also be zero).

The class provides a method `buyNewCopies` for resupplying the available games in the shop, and a public method `sellGames` that tries to satisfy a customer's order with the available copies of games.

After carefully reading the code, you have to answer the following questions. **Notice that each answer requires you to write a single value!** Please make sure to mark the answers clearly on the sheets. Moreover, **you have to develop your solution by motivating your answers on the sheets** (that you need to upload if you are attending this exam online). **Only providing the final answers is not enough for obtaining points in this section.**

1. What is the value assigned to `r1` at line 36?
2. Is the message defined in line 40 printed at runtime?
3. How many games are still available in `s3` (sum of values in stock) after line 55?
4. What value is returned at line 56?

Provided source code:

- **Shop.h**

```
#include <memory>
#include <string>
#include <unordered_set>
#include <unordered_map>

enum Item {Monopoly, Risiko, Catan};

typedef std::shared_ptr<std::unordered_set<Item>> availableGames;
typedef std::unordered_map<Item, unsigned int> orderType;

class Shop {
private:
  std::string ID;
  availableGames list;
  std::unordered_map<Item, unsigned int> availableCopies;

  void buyNewCopies();

public:
  Shop(std::string name, availableGames &c) : ID(name), list(c) {
    buyNewCopies();
  }

  Shop(const Shop &s) : ID(s.ID + "_copy"), availableCopies(s.availableCopies), list(s.list) {
     buyNewCopies(); };

  Shop() : ID("noname") {};

  Shop& operator=(const Shop &s);

  orderType order(orderType shoppingCart);

};
```

- **Shop.cpp**

```
#include "Shop.h"
#include <unordered_map>

void Shop::buyNewCopies() {
  for (const Item &it : *list) {
    availableCopies[it] = 5;
  }
}

Shop& Shop::operator=(const Shop &s) {
  list = s.list;
  return *this;
}

orderType Shop::order(orderType shoppingCart) {
  orderType retList;
  for (auto &order : shoppingCart) {
    if (availableCopies[order.first] >= order.second) {
      availableCopies[order.first] = availableCopies[order.first] - order.second;
      retList[order.first] = 0;
    } else {
      retList[order.first] = order.second - availableCopies[order.first];
      availableCopies[order.first] = 0;
    }
  }
  return retList;
}
```

- **main.cpp**

```
#include "Shop.h"
#include <iostream>
#include <memory>
#include <type_traits>

void createCatalogue(availableGames &cat) {
  cat->insert(Item::Monopoly);
  cat->insert(Item::Risiko);
}

bool sellGames(orderType order, Shop &s) {
  bool retVal = true;

  orderType result = s.order(order);

  for (auto &element : result) {
    retVal = retVal && element.second == 0;
  }

  return retVal;
}

int main() {

  availableGames cat1 = std::make_shared<std::unordered_set<Item>>();
  createCatalogue(cat1);

  Shop s1("Main shop", cat1);
  Shop s2;
  s2 = s1;

  orderType order1;
  order1[Item::Monopoly] = 3;
  order1[Item::Catan] = 2;

  bool r1 = sellGames(order1, s1); std::cout << "Value of r1: " << r1 << std::endl;
  bool r2 = sellGames(order1, s2);

  if (r1 && r2) {
    std::cout << "Sold out!" << std::endl;
  }

  availableGames cat2 = cat1;
  cat2->insert(Item::Risiko);
  cat2->erase(Item::Catan);

  Shop s3(s1);

  orderType order2;
  order2[Item::Risiko] = 2;
  order2[Item::Catan] = 1;

  Shop s4("Downtown shop", cat2);

  bool r3 = sellGames(order2, s3) && sellGames(order2, s4);
  return r3;

}
```

Solution 3

1. The assigned value is 0. In fact, the order is asking for copies of Catan which are not available since they were not allocated using the constructor.
2. The line is not printed. The value of `r2` is zero because `s2` is create with a different constructor and the assignment operator does not add any resource to the stock of `s2`. Furthermore, `r1` equals 0.
3. The answer is 8. In fact, `s3` copies the same resources present in `s1` and then calls `buyNewCopies`.
4. The returned value is 0. In fact, `s4` is resupplied after creation but has no copies of Catan to fulfil the order.