Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna, Prof. Damian Tamburri

Date: 05-09-2024

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)

PoliMi has initiated a new collaboration with the National Astronomical Observatory and requires you to implement a planetarium management system—PoliStars—where you can manage celestial bodies and their relationships while they move on a 2D plane. In particular, the trajectory of a celestial body is a collection of 2D coordinates, each representing its position at a certain time instant. You can assume these collections to all have the same length since they cover a fixed time window. Moreover, a planet and a star shall be linked if the planet orbits the star. Use shared pointers to manage the memory of the celestial bodies. Use STL containers to store the stars and planets. The system is structured as follows:

- the `CelestialBody` base class represents a generic celestial body with virtual methods for displaying and getting the name (i.e., a `std::string`), mass (i.e., a `double`), and trajectory (i.e., a vector of pairs of `double`s);
- `Star` inherits from `CelestialBody`, includes the star type, and has a method to add planets orbiting it;
- the `Planet` class also inherits from `CelestialBody`;
- the `Planetarium` class manages collections of stars and planets, and includes methods for adding stars and planets, linking planets to stars, and displaying characteristics of any celestial body.

Fig. 1 provides a class diagram for the aforementioned system. For the sake of simplicity, **the diagram does not include `getters` and not `setters`, but both are available for you to use** if needed.
Given the specifications above, your task is to implement:

1. the `Planetarium` default constructor which initializes the data structures used to store the stars and planets. Specifically, it initializes the vectors `stars` and `planets` to store the shared pointers to `Star` and `Planet` objects, respectively. It also shall initialize the unordered maps `starMap` and `planetMap` to map the names of stars and planets to their corresponding `shared_pointer` objects for quicker lookup;
2. the `void Star::displayPlanets() const` method, whose purpose is to display information about all the planets that are orbiting the star;
3. the `void Planetarium::calculateGravitationalForces() const` method that calculates and visually displays the mutual gravitational forces between every pair of celestial bodies (stars and planets) in the planetarium, in their initial positions. The calculation is based on Newton's law of universal gravitation:

$$F = G\frac{m_1 m_2}{r^2}$$

where:

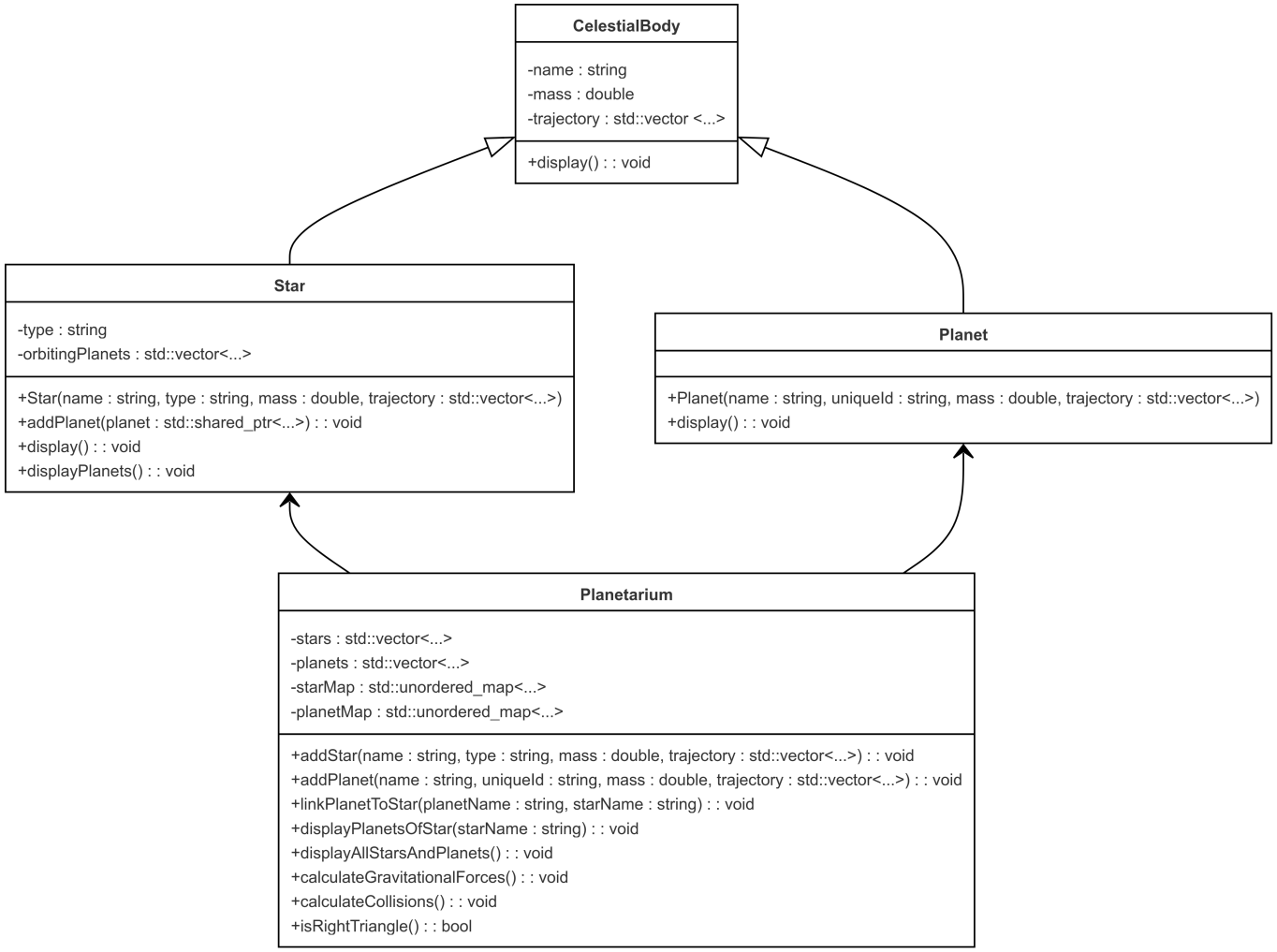

Figure 1: a simplistic class diagram for PoliStars. **NOTE:** angular brackets template content definitions are deliberately left blank for you to resolve.

- F is the gravitational force between two masses;
- G is the gravitational constant, defined as **global** `const double G` and therefore available to you at any point of the code;
- m_1 and m_2 are the masses of the two bodies;
- r is the distance between the centers of the two masses.

4. the `Planetarium::isRightTriangle()` method to calculate the right triangle problem given three celestial bodies, passed as arguments via `shared_ptr`s. The method returns whether or not the initial positions of three bodies form a right triangle;
5. the worst-case computational complexity of all methods you have implemented;
6. finally, discuss at least one opportunity to improve the computational complexity of the `calculateGravitationalForces` method.

Solution 1

1. A possible implementation of the `Planetarium` default constructor is listed below:

```
1    // Constructor
2    Planetarium() {
```

```
        stars = std::vector<std::shared_ptr<Star>>();
        planets = std::vector<std::shared_ptr<Planet>>();
        starMap = std::unordered_map<std::string, std::shared_ptr<Star>>();
        planetMap = std::unordered_map<std::string, std::shared_ptr<Planet>>();
    }
```

2. The displayPlanets method is used to display contents of the planets and their specifics:

```
    // Function to display all planets orbiting the star
    void displayPlanets() const {
        if (orbitingPlanets.empty()) {
            std::cout << "No planets orbiting " << name << ".\n";
            return;
        }
        std::cout << "Planets orbiting " << name << ":\n";
        for (const auto &planet : orbitingPlanets) {
            planet->display();
        }
    }
```

3. The calculateGravitationalForces method, as per specification, iterates through all pairs of celestial bodies, computes the distance between them, and then calculates and prints the gravitational force. Specifically:

```
    // Function to calculate mutual gravitational forces between celestial bodies
    void calculateGravitationalForces() const {
        std::vector<std::shared_ptr<CelestialBody>> bodies;
        bodies.insert(bodies.end(), stars.begin(), stars.end());
        bodies.insert(bodies.end(), planets.begin(), planets.end());

        // Iterate through all pairs of celestial bodies
        for (size_t i = 0; i < bodies.size(); ++i) {
            for (size_t j = i + 1; j < bodies.size(); ++j) {
                auto body1 = bodies[i];
                auto body2 = bodies[j];

                double mass1 = body1->getMass();
                double mass2 = body2->getMass();
                auto pos1 = body1->getTrajectory().front();
                auto pos2 = body2->getTrajectory().front();

                double dx = pos2.first - pos1.first;
                double dy = pos2.second - pos1.second;
                double distance = std::sqrt(dx * dx + dy * dy);

                double force = G * ((mass1 * mass2) / (distance * distance));

                std::cout << "Gravitational force between " << body1->getName()
                          << " and " << body2->getName() << " is " << force << " N" << std::endl;
            }
        }
    }
```

4. In the isRightTriangle method, the goal is to determine if three celestial bodies form a right triangle by checking if any combination of their squared distances satisfies the Pythagorean theorem; specifically, the method calculates the squared distances between each pair of bodies. Subsequently, it checks if any combination of these distances satisfies the Pythagorean theorem returning **true** if the bodies form a right triangle and **false** otherwise. Specifically:

```
    // Method to check if three celestial bodies form a right triangle
```

```
bool isRightTriangle(const std::shared_ptr<CelestialBody> &body1, const std::shared_ptr<
  CelestialBody> &body2, const std::shared_ptr<CelestialBody> &body3) const {
    auto pos1 = body1->getTrajectory().front();
    auto pos2 = body2->getTrajectory().front();
    auto pos3 = body3->getTrajectory().front();

    double d12 = std::pow(pos2.first - pos1.first, 2) + std::pow(pos2.second - pos1.second, 2);
    double d23 = std::pow(pos3.first - pos2.first, 2) + std::pow(pos3.second - pos2.second, 2);
    double d31 = std::pow(pos3.first - pos1.first, 2) + std::pow(pos3.second - pos1.second, 2);

    // Check if any combination of the sides forms a right triangle
    return (d12 + d23 == d31) || (d23 + d31 == d12) || (d31 + d12 == d23);
}
```

5. We discuss the worst-case computational complexity of the methods implemented above. The `Planetarium` default constructor bears constant computational complexity ($O(1)$) as it merely initializes two vectors—for stars and planets—as well as two maps, again for stars and planets.

   Concerning `displayPlanets`, in the worst case, we must assume that all planets in the planetarium orbit the star under scrutiny. The method iterates over the `orbitingPlanets` vector and calls `display` on each planet, where p is the number of planets. Therefore the complexity in the worst-case is $O(p)$.

   Secondly, concerning `calculateGravitationalForces`, it mainly compares all pairs of celestial bodies (both stars and planets) thus amounting to a complexity of $O((s+p)^2)$ because it involves a double loop over all bodies, where s is the number of stars.

   Finally, `isRightTriangle` has $O(1)$ computational complexity since it only involves a constant number of algebraic operations.

6. Opportunities to improve the `calculateGravitationalForces` method could be:

   - first, the algorithm could feature some kind of *spatial partitioning* approach to limit calculations only to nearby bodies; for example a spatial partitioning based on a grid can significantly reduce the computational load by dividing the simulation space into smaller regions, which limits the number of interactions that need to be calculated. In a typical $O((s+p)^2)$ algorithm, where every pair of celestial bodies is considered, the computational burden becomes significant as the number of bodies increases. Ideally, spatial partitioning mitigates this by leveraging the fact that gravitational forces are strongest between nearby bodies and weaker (and sometimes negligible) between distant ones, calculations for these may therefore be omitted;
   - secondly, an opportunistic way to improve the method could be to use parallel computing techniques to divide the work of force calculations across multiple processors. For example, by distributing the pairwise calculations among multiple processors, the overall time required can be reduced significantly. In that instance, the computational time can be reduced by a factor of $1/P$, where P is the number of processors.

Exercise 2 (13 points)

Natural Language Processing (NLP) is a field of computer science concerned with providing computers the ability to process data encoded in natural language, which is any language that is spoken or written naturally in human interactions. A key idea of modern deep learning approaches to NLP is *word embeddings*, which are a way of representing words. Specifically, we choose a set of d *features*, each of which represents a specific concept, such as "Gender" and "Royalty". Then, we represent each word as a d-dimensional vector that encodes the relation between the word and each feature. In our case, the vector entries are `double`s between -1 and 1, and the larger the absolute value is, the stronger the correlation with the feature. Given a word w, we will denote its embedding as e_w.

	Gender	**Royalty**	**Age**	**Food**
man	-1	0.01	0.03	0.04
woman	1	0.02	0.02	0.01
king	-0.95	0.93	0.70	0.02
queen	0.97	0.95	0.69	0.01
apple	0.00	-0.01	0.03	0.95

Table 1: Example of an embedding matrix with dimension $d = 4$. Rows are words and columns are features.

As an example, Table 1 represents a *corpus*, i.e., a matrix of word embeddings representing all the known words in a language. The words "man" and "woman" are strongly correlated to the concept of "Gender" and sit at the extremes of the spectrum, having values of -1 and 1, respectively. "King" also correlates with "Gender" since it implies maleness. Instead, "man" and "woman" have low "Royalty" scores since they are poorly correlated with the concept, whereas "king" has a high score. Similar arguments hold for "queen" and "apple".

Representing words in a vector space allows one to define similarity scores between them. An example of such a score is the *cosine similarity*, which we will denote as $sim(e_a, e_b)$: the larger its value, the more similar a and b are.

NLP is also interested in learning analogies between words, for instance "man is to woman as king is to queen". Given a corpus W and three words a, b, c, one seeks to find the word $x \in W$ that is the most fitting for the analogy "a is to b as c is to x" (in the above example, x would be "queen"). This means that the difference between a and b and the one between c and x must be similar to each other. From a formal perspective, we can find word x and its corresponding score s^* by solving the following problem:

$$s^* = \max_{w \in W} sim(e_w, e_c - e_a + e_b). \tag{1}$$

Under the aforementioned definitions, you have to:

1. implement a parallel function with the following signature:
   `double compute_max_sim(const std::string &corpus_file, const std::string &ABC_file)`
   that returns the maximum similarity value found in the corpus, i.e., the s^* in Equation 1. The two arguments are the names of the files containing, respectively, the embedding matrix of the entire corpus W and the embedding matrix of the words a, b, and c (in this order). The **file contents must only be read by rank 0**. You can assume that **the number of rows in the corpus is a multiple of the number of parallel processes**.

2. complete the skeleton of the `main.cpp` file below in which you read the two file names from the command-line arguments of `int main(int argc, char* argv[ ])`, invoke the `compute_max_sim` function appropriately, and print (with rank 0 only) the maximum found. You must handle command-line arguments correctly and **display an error message** if the user provides an incorrect number of arguments.

For your implementation, you can rely on the following function:
`double cos_sim(const la::dense_matrix &a, const la::dense_matrix &b)`
that returns the cosine similarity between row vectors `a` and `b`, and is already implemented and available for you to use. Moreover, the matrices are represented using the `la::dense_matrix` class, whose declaration is reported below. Note that **we define additional methods with respect to the usual `dense_matrix`**: namely, `operator+` and `operator-`, two external functions that perform the corresponding element-wise operations, and `dense_matrix::row (size_t index) const`, which returns the matrix row corresponding to the given `index`.

- **main.cpp:**

```
#include <iostream>
#include <fstream>
```

```
#include <sstream>
#include <string>
#include <mpi.h>

#include "dense_matrix_plus.hpp"


double dot(const la::dense_matrix &a, const la::dense_matrix &b);
double cos_sim(const la::dense_matrix &a, const la::dense_matrix &b);
double compute_max_sim(const std::string &corpus_file,
                       const std::string &ABC_file);


int main(int argc, char* argv[ ])  {
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // YOUR CODE GOES HERE

  MPI_Finalize();
  return 0;
}
```

- **dense_matrix.hpp:**

```
#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
    row (size_t) const;
  };

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

  void
  swap (dense_matrix &, dense_matrix &);

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

  dense_matrix
  operator - (dense_matrix const &, dense_matrix const &);
}

#endif // DENSE_MATRIX_HH
```

Solution 2

An example solution is reported below.

```
1 double compute_max_sim(const std::string &corpus_file, const std::string &ABC_file) {
```

```
2   int rank, size;
3   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4   MPI_Comm_size(MPI_COMM_WORLD, &size);
5
6   la::dense_matrix corpus, ABC;
7
8   if (rank == 0) {
9     // Load matrices (rows are words, columns are features)
10    std::ifstream ifs1(corpus_file), ifs2(ABC_file);
11    corpus.read(ifs1);
12    ABC.read(ifs2);
13    ifs1.close();
14    ifs2.close();
15
16    // Print matrices
17    std::cout << "Corpus:" << std::endl;
18    corpus.print(std::cout);
19    std::cout << std::endl << "ABC:" << std::endl;
20    ABC.print(std::cout);
21    }
22
23  // Broadcast ABC matrix to other ranks
24
25  // Brodcast ABC matrix number of rows and columns
26  unsigned m = ABC.rows(), n = ABC.columns();
27  MPI_Bcast(&m, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
28  MPI_Bcast(&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
29
30  if (rank != 0) // resize ABC matrix in other ranks
31    ABC = la::dense_matrix(m, n);
32
33  MPI_Bcast(ABC.data(), m*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
34
35  // Brodcast corpus matrix number of rows and columns
36  m = corpus.rows();
37  n = corpus.columns();
38  MPI_Bcast(&m, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
39  MPI_Bcast(&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
40
41  const unsigned stripe = m / size;
42  la::dense_matrix local_corpus(stripe, n);
43
44  // Scatter corpus matrix
45  MPI_Scatter(corpus.data(), stripe*n, MPI_DOUBLE,
46            local_corpus.data(), stripe*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
47
48  // Get first three terms of proportion: A : B = C : x
49  la::dense_matrix A = ABC.row(0);
50  la::dense_matrix B = ABC.row(1);
51  la::dense_matrix C = ABC.row(2);
52
53  // Store comparison term
54  la::dense_matrix rhs = C - A + B;
55  // Loop to find x
56  double max_sim = 0.0;
57  size_t max_idx = 0;
58
59  // find local max similarity
```

```
  for (size_t i = 0; i < local_corpus.rows(); i++) {
    double sim = cos_sim(local_corpus.row(i), rhs);
    if (sim > max_sim) {
      max_idx = i;
      max_sim = sim;
    }
  }

  MPI_Allreduce(MPI_IN_PLACE, &max_sim, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

  return max_sim;
}
```

The code is based on the following main steps:

- The root process (rank 0) loads the corpus and `ABC` matrices from the specified files. It also prints these matrices for verification purposes.
- The dimensions of the `ABC` matrix (`m` rows and `n` columns) are broadcast to all processes. Non-root processes (those with ranks other than 0) resize their local `ABC` matrix based on the received dimensions. The actual data of the `ABC` matrix is then broadcast to all processes. Since the `ABC` matrix includes the embedding of only three words, the broadcast operation is not too expensive.
- The root process broadcasts the dimensions of the corpus matrix to all processes. The corpus matrix is then divided into stripes, where each process gets a subset of rows (a "stripe") to work on.
- The first three rows of the `ABC` matrix are extracted as `A`, `B`, and `C`. The reference vector `rhs` is computed as `C - A + B`. This vector will be compared with each row of the corpus matrix.
- Each process computes the cosine similarity between each row of its local corpus matrix and the reference vector `rhs`. It keeps track of the maximum similarity (`max_sim`) and the corresponding row index (`max_idx`) within its local data (this latter operation is optional).
- `MPI_Allreduce` is used to find the maximum similarity (`max_sim`) across all processes. The result is stored back in `max_sim`, and now every process has the global maximum similarity value, which is finally returned to the main.

A possible implementation of the main function is the following:

```
int main(int argc, char* argv[ ]) {
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (argc != 3){
    std::cout << "The number of parameters is not correct" << std::endl;
    MPI_Finalize();
    return -1;
  }

  double max_sim = compute_max_sim(argv[1], argv[2]);

  if (rank == 0)
    std::cout << "Best match: " << max_sim << std::endl;

  MPI_Finalize();
  return 0;
}
```

The program expects exactly two command-line arguments (in addition to the program name), typically file paths. If the number of arguments is incorrect, it prints an error message and terminates the program using `MPI_Finalize` to clean up the MPI environment. Otherwise, the `compute_max_sim` function is called with the two command-line arguments, `argv[1]` and `argv[2]`, which are expected to be file paths to the corpus and ABC matrices, respectively. The `compute_max_sim` finally performs the main computation (as explained above).

Exercise 3 (5 points)

Consider the five snippets of C++ code that follow. Every individual snippet **may or may not contain** at least one specific problem or contain provocative bugs within them. Study each snippet and identify the issue in it, if any. Specifically, observe each line and comment on the bugs you identify, their type, and their effects on the program execution, such as crashes, unexpected output, or memory allocation problems.

Algorithm 1

```
#include <iostream>

int main() {
    int* ptr;

    *ptr = 10;

    std::cout << *ptr << std::endl;

    return 0;
}
```

Algorithm 2

```
#include <iostream>

int main() {
    int* arr = new int[10];

    return 0;
}
```

Algorithm 3

```
#include <iostream>

int main() {
    int* ptr = new int(5);
    delete ptr;

    *ptr = 10;

    std::cout << *ptr << std::endl;

    return 0;
}
```

Algorithm 4

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2 = vec1; // Copying vec1 to vec2

    vec1.push_back(4); // Modifying vec1

    std::cout << "vec1 size: " << vec1.size() << std::endl; // Accessing vec1 after modification
    std::cout << "vec2 size: " << vec2.size() << std::endl; // Accessing vec2 after modification to vec1

    return 0;
}
```

Algorithm 5

```cpp
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass(int val) : value(val) {}
    void print() {
        std::cout << "Value: " << value << std::endl;
    }
private:
    int value;
};

int main() {
    std::shared_ptr<MyClass> ptr1(new MyClass(10));

    std::shared_ptr<MyClass> ptr2 = ptr1;

    if (ptr1) {
        ptr1.reset();
    }

    ptr2->print();

    return 0;
}
```

Solution 3

1. The pointer `ptr` is declared but not initialized, meaning it could point to any random memory location. Dereferencing an uninitialized pointer leads to undefined behavior, which can cause a program crash or other unpredictable behavior.

2. The program allocates memory for an array but does not deallocate it using **`delete`**. This results in a memory leak. Not freeing allocated memory can lead to increased memory usage over time, potentially causing the program to run out of memory.

3. After calling **delete** on ptr, it becomes a dangling pointer, meaning it points to memory that has been freed. Dereferencing a dangling pointer leads to undefined behavior, similar to dereferencing an uninitialized pointer.

4. This snippet contains no errors, although it demonstrates potential confusion when copying vectors. Modifying vec1 after copying to vec2 does not affect vec2, but it might be misinterpreted if one expects changes in vec1 to reflect in vec2.

5. Using **new** instead of std::make_shared is the problem; line 15 directly uses **new** to allocate memory for the MyClass object and then wraps it in a std::shared_ptr. A better approach would be to use std::make_shared <MyClass> instead. This way is more efficient and safer because it allocates both the object and the control block in a single memory allocation and also provides better exception safety.