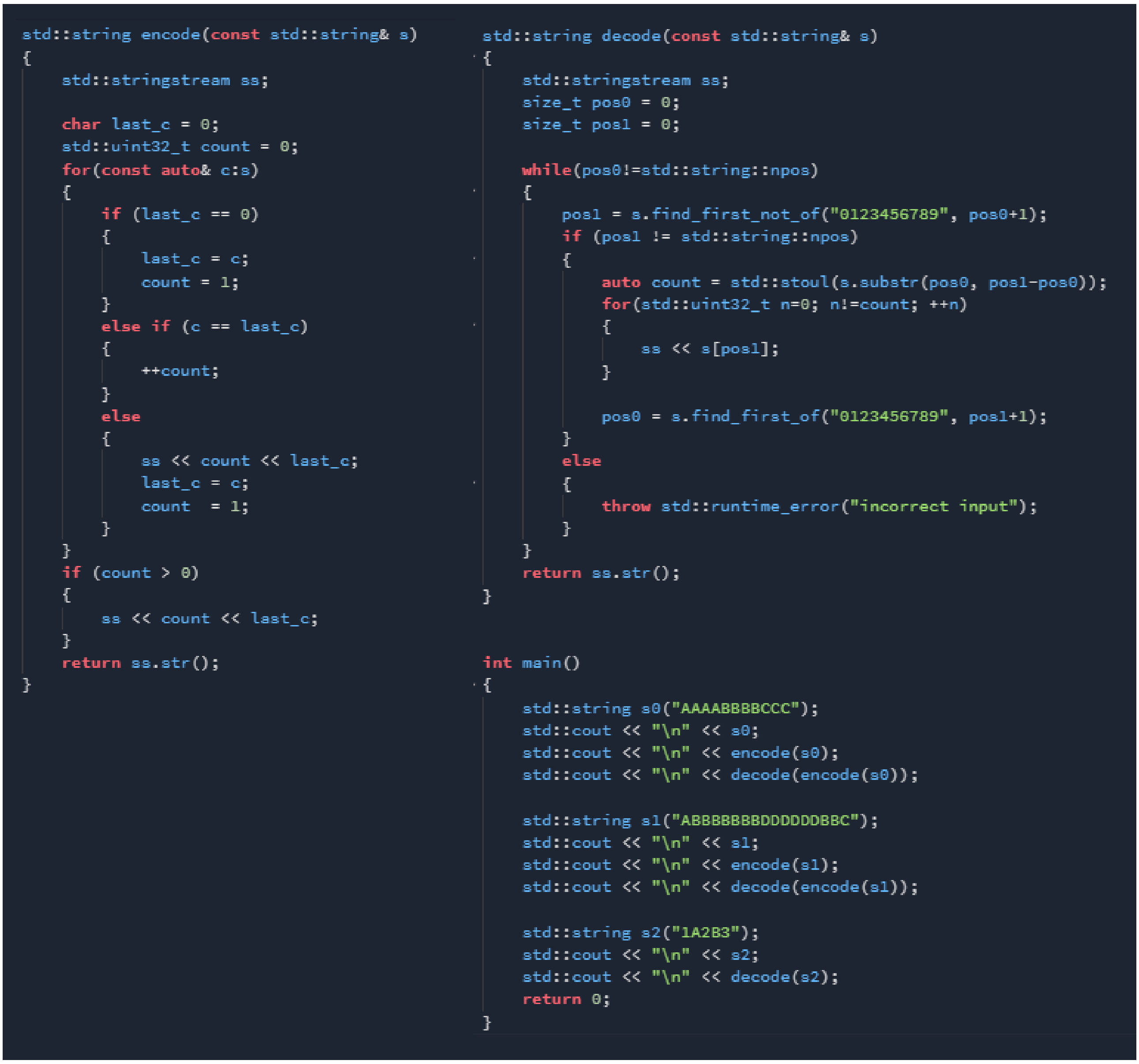
Square Point UK

1st round - 7 June 2021

Implement run-length encoding and decoding

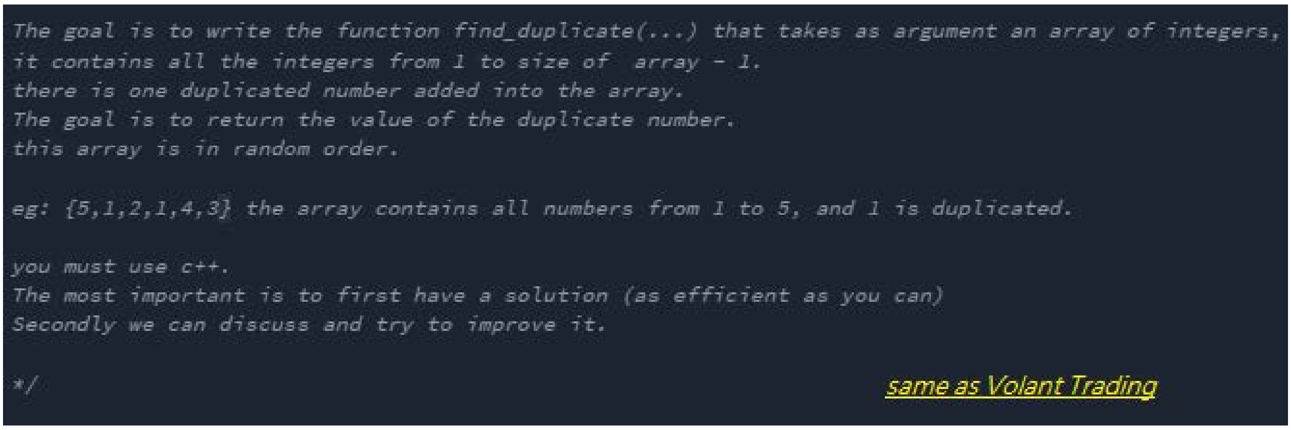


```cpp
std::string encode(const std::string& s)
{
    std::stringstream ss;

    char last_c = 0;
    std::uint32_t count = 0;
    for(const auto& c:s)
    {
        if (last_c == 0)
        {
            last_c = c;
            count = 1;
        }
        else if (c == last_c)
        {
            ++count;
        }
        else
        {
            ss << count << last_c;
            last_c = c;
            count  = 1;
        }
    }
    if (count > 0)
    {
        ss << count << last_c;
    }
    return ss.str();
}
```

```cpp
std::string decode(const std::string& s)
{
    std::stringstream ss;
    size_t pos0 = 0;
    size_t pos1 = 0;

    while(pos0!=std::string::npos)
    {
        pos1 = s.find_first_not_of("0123456789", pos0+1);
        if (pos1 != std::string::npos)
        {
            auto count = std::stoul(s.substr(pos0, pos1-pos0));
            for(std::uint32_t n=0; n!=count; ++n)
            {
                ss << s[pos1];
            }

            pos0 = s.find_first_of("0123456789", pos1+1);
        }
        else
        {
            throw std::runtime_error("incorrect input");
        }
    }
    return ss.str();
}


int main()
{
    std::string s0("AAAABBBBCCC");
    std::cout << "\n" << s0;
    std::cout << "\n" << encode(s0);
    std::cout << "\n" << decode(encode(s0));

    std::string s1("ABBBBBBBDDDDDDBBC");
    std::cout << "\n" << s1;
    std::cout << "\n" << encode(s1);
    std::cout << "\n" << decode(encode(s1));

    std::string s2("1A2B3");
    std::cout << "\n" << s2;
    std::cout << "\n" << decode(s2);
    return 0;
}
```

2nd round - 16 June 2021

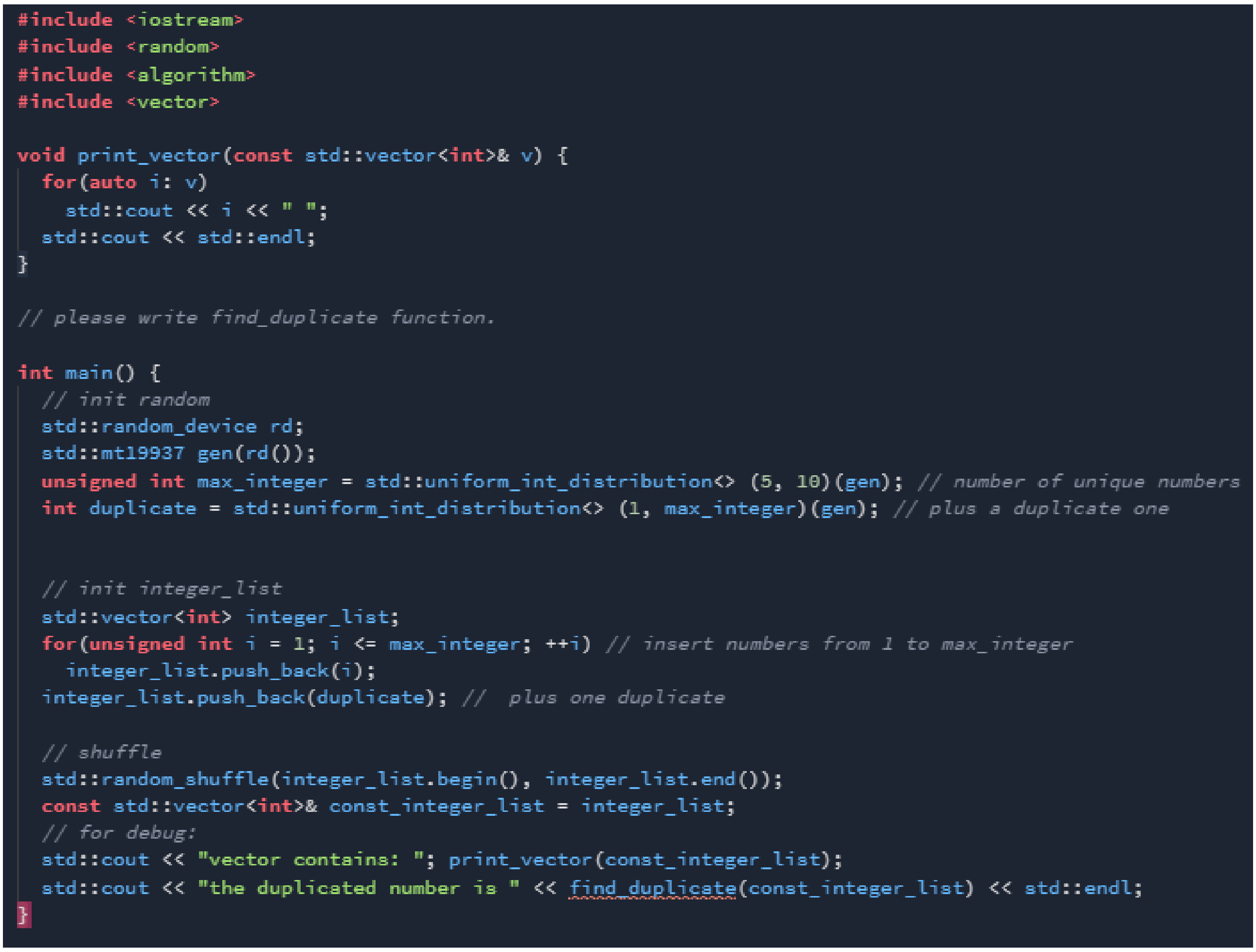
Question 1 : Implement the `find_duplicate` function below.



```
The goal is to write the function find_duplicate(...) that takes as argument an array of integers,
it contains all the integers from 1 to size of  array - 1.
there is one duplicated number added into the array.
The goal is to return the value of the duplicate number.
this array is in random order.

eg: {5,1,2,1,4,3} the array contains all numbers from 1 to 5, and 1 is duplicated.

you must use c++.
The most important is to first have a solution (as efficient as you can)
Secondly we can discuss and try to improve it.

*/
```

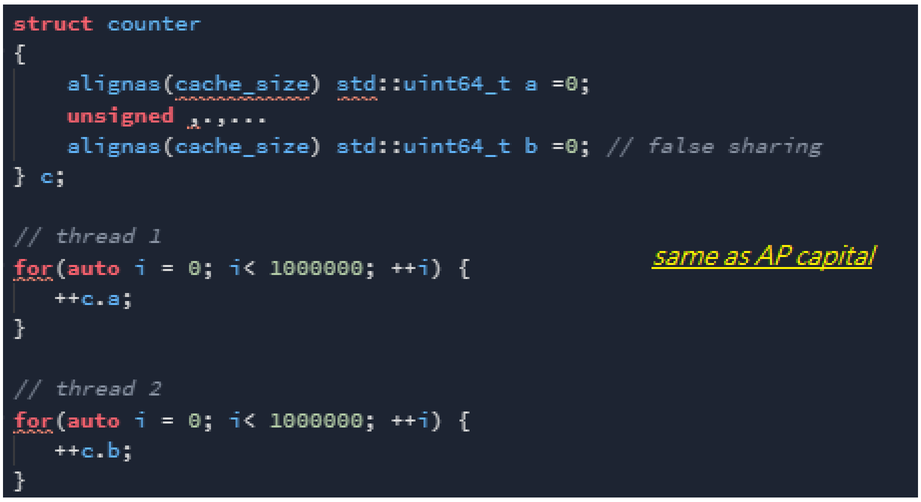
same as Volant Trading

Here is the solution :



```
#include <iostream>
#include <random>
#include <algorithm>
#include <vector>

void print_vector(const std::vector<int>& v) {
  for(auto i: v)
    std::cout << i << " ";
  std::cout << std::endl;
}

// please write find_duplicate function.

int main() {
  // init random
  std::random_device rd;
  std::mt19937 gen(rd());
  unsigned int max_integer = std::uniform_int_distribution<> (5, 10)(gen); // number of unique numbers
  int duplicate = std::uniform_int_distribution<> (1, max_integer)(gen); // plus a duplicate one


  // init integer_list
  std::vector<int> integer_list;
  for(unsigned int i = 1; i <= max_integer; ++i) // insert numbers from 1 to max_integer
    integer_list.push_back(i);
  integer_list.push_back(duplicate); //  plus one duplicate

  // shuffle
  std::random_shuffle(integer_list.begin(), integer_list.end());
  const std::vector<int>& const_integer_list = integer_list;
  // for debug:
  std::cout << "vector contains: "; print_vector(const_integer_list);
  std::cout << "the duplicated number is " << find_duplicate(const_integer_list) << std::endl;
}
```

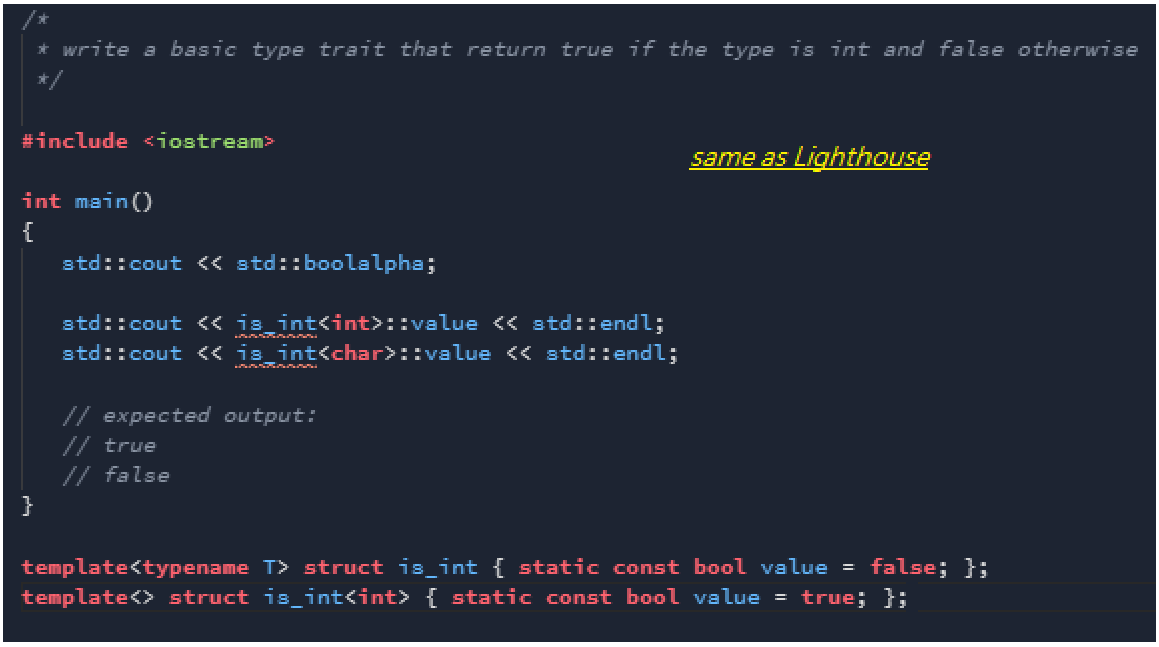
Question 2 : What is the problem with the following code?



```
struct counter
{
    alignas(cache_size) std::uint64_t a =0;
    unsigned a.;...
    alignas(cache_size) std::uint64_t b =0; // false sharing
} c;

// thread 1
for(auto i = 0; i< 1000000; ++i) {
    ++c.a;
}

// thread 2
for(auto i = 0; i< 1000000; ++i) {
    ++c.b;
}
```

same as AP capital

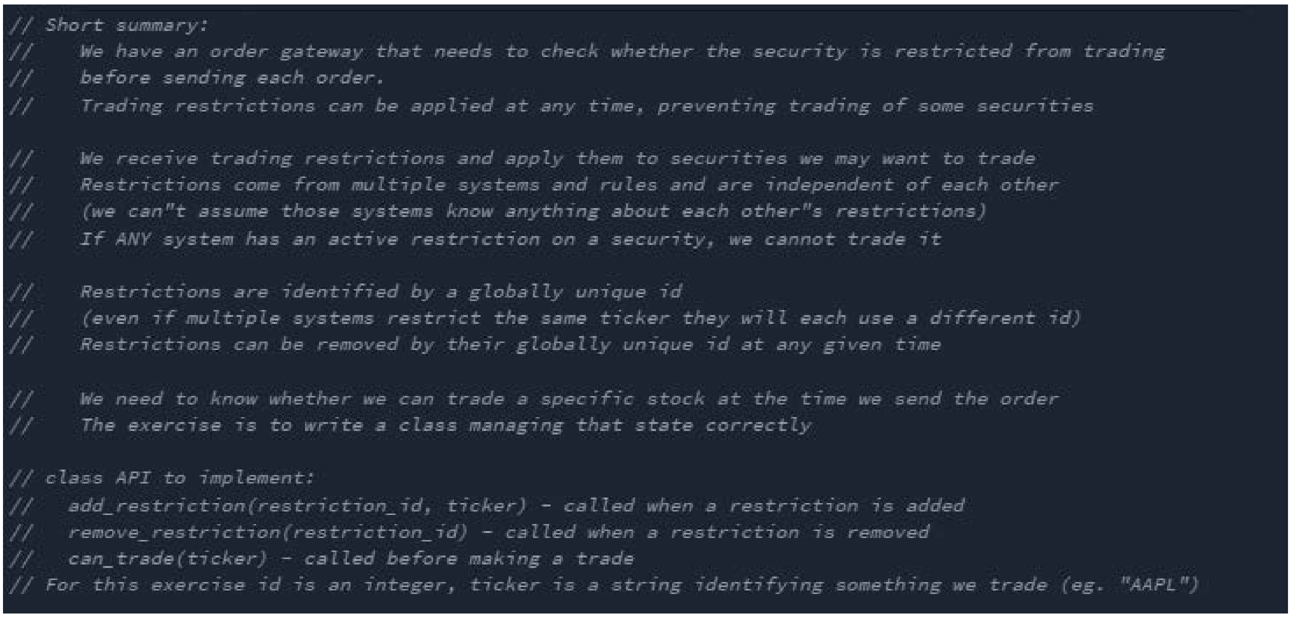
Question 3 : Implement the template traits.



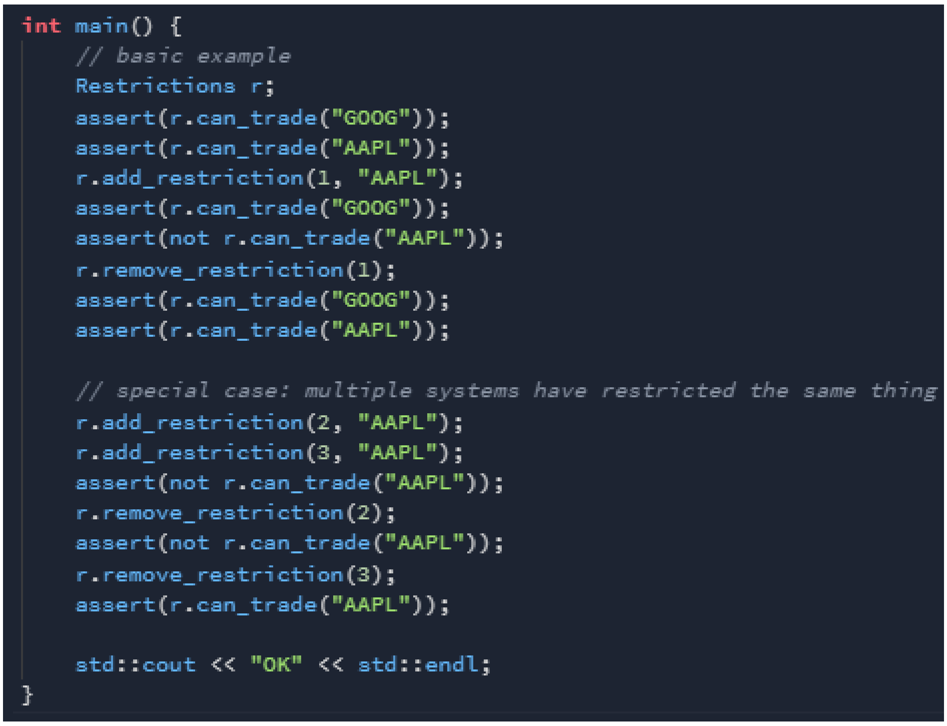
```
/*
 * write a basic type trait that return true if the type is int and false otherwise
 */

#include <iostream>

int main()
{
   std::cout << std::boolalpha;

   std::cout << is_int<int>::value << std::endl;
   std::cout << is_int<char>::value << std::endl;

   // expected output:
   // true
   // false
}

template<typename T> struct is_int { static const bool value = false; };
template<> struct is_int<int> { static const bool value = true; };
```

same as Lighthouse

3rd round - 28 June 2021
Implement the required class (Hint : two-containers approach)

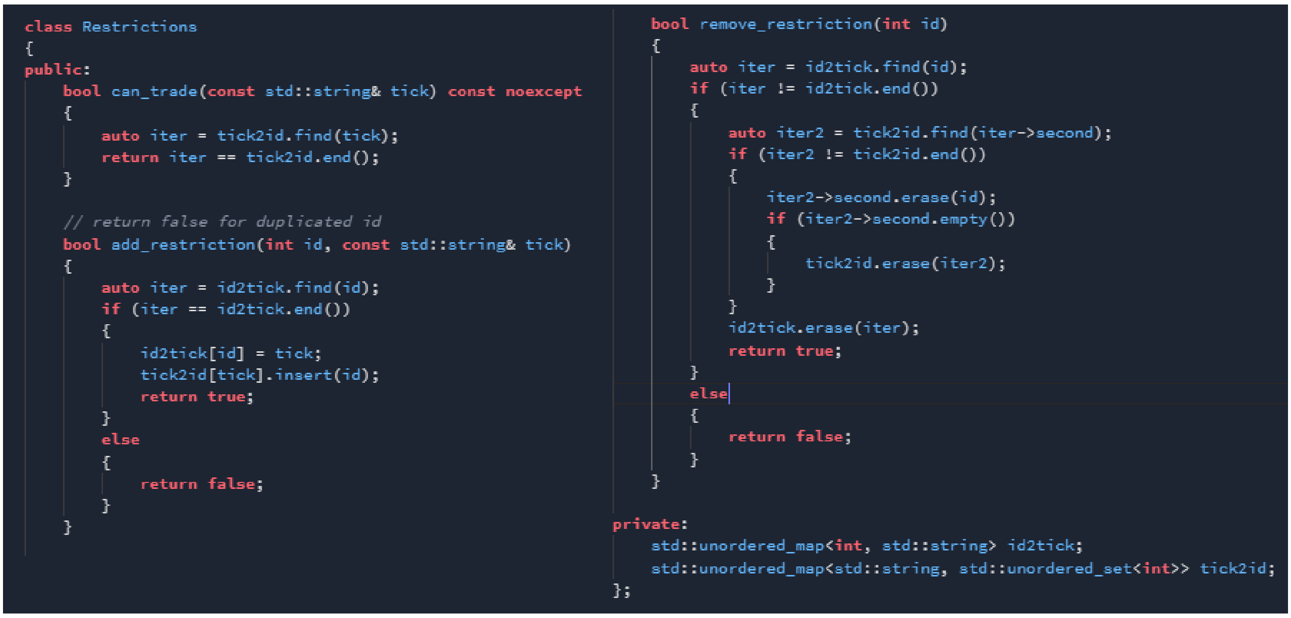


```
// Short summary:
//    We have an order gateway that needs to check whether the security is restricted from trading
//    before sending each order.
//    Trading restrictions can be applied at any time, preventing trading of some securities

//    We receive trading restrictions and apply them to securities we may want to trade
//    Restrictions come from multiple systems and rules and are independent of each other
//    (we can"t assume those systems know anything about each other"s restrictions)
//    If ANY system has an active restriction on a security, we cannot trade it

//    Restrictions are identified by a globally unique id
//    (even if multiple systems restrict the same ticker they will each use a different id)
//    Restrictions can be removed by their globally unique id at any given time

//    We need to know whether we can trade a specific stock at the time we send the order
//    The exercise is to write a class managing that state correctly

// class API to implement:
//   add_restriction(restriction_id, ticker) - called when a restriction is added
//   remove_restriction(restriction_id) - called when a restriction is removed
//   can_trade(ticker) - called before making a trade
// For this exercise id is an integer, ticker is a string identifying something we trade (eg. "AAPL")
```



```cpp
int main() {
    // basic example
    Restrictions r;
    assert(r.can_trade("GOOG"));
    assert(r.can_trade("AAPL"));
    r.add_restriction(1, "AAPL");
    assert(r.can_trade("GOOG"));
    assert(not r.can_trade("AAPL"));
    r.remove_restriction(1);
    assert(r.can_trade("GOOG"));
    assert(r.can_trade("AAPL"));

    // special case: multiple systems have restricted the same thing
    r.add_restriction(2, "AAPL");
    r.add_restriction(3, "AAPL");
    assert(not r.can_trade("AAPL"));
    r.remove_restriction(2);
    assert(not r.can_trade("AAPL"));
    r.remove_restriction(3);
    assert(r.can_trade("AAPL"));

    std::cout << "OK" << std::endl;
}
```

Here is the solution :



```cpp
class Restrictions
{
public:
    bool can_trade(const std::string& tick) const noexcept
    {
        auto iter = tick2id.find(tick);
        return iter == tick2id.end();
    }

    // return false for duplicated id
    bool add_restriction(int id, const std::string& tick)
    {
        auto iter = id2tick.find(id);
        if (iter == id2tick.end())
        {
            id2tick[id] = tick;
            tick2id[tick].insert(id);
            return true;
        }
        else
        {
            return false;
        }
    }
    bool remove_restriction(int id)
    {
        auto iter = id2tick.find(id);
        if (iter != id2tick.end())
        {
            auto iter2 = tick2id.find(iter->second);
            if (iter2 != tick2id.end())
            {
                iter2->second.erase(id);
                if (iter2->second.empty())
                {
                    tick2id.erase(iter2);
                }
            }
            id2tick.erase(iter);
            return true;
        }
        else
        {
            return false;
        }
    }

private:
    std::unordered_map<int, std::string> id2tick;
    std::unordered_map<std::string, std::unordered_set<int>> tick2id;
};
```