```
#ifndef CODEREVIEWTASK_MYVECTOR_HPP
#define CODEREVIEWTASK_MYVECTOR_HPP
#include <vector>
#include <string>
#include <algorithm>
#include <stdexcept>

/*
 * MyVector stores a collection of objects with their names.
 *
 * For each object T, MyVector stores T`s name as std::string.
 * Several objects can have similar name.
 * operator[](const std::string& name) should return the first object
 * with the given name.
 *
 * Your task is to find as many mistakes and drawbacks in this code
 * (according to the presentation) as you can.
 * Annotate these mistakes with comments.
 *
 * Once you have found all the mistakes, rewrite the code
 * so it would not change its original purpose
 * and it would contain no mistakes.
 * Try to make the code more efficient without premature optimization.
 *
 * You can change MyVector interface completely, but there are several rules:
 * 1) you should correctly and fully implement copy-on-write idiom.
 * 2) std::pair<const T&, const std::string&> operator[](int index) const must take constant time
at worst.
 * 3) const T& operator[](const std::string& name) const should be present.
 * 4) both operator[] should have non-const version.
 * 5) your implementation should provide all the member types of std::vector.
 * 6) your implementation should provide the following functions:
 * 1) begin(), cbegin(), end(), cend()
 * 2) empty(), size()
 * 3) reserve(), clear()
 */

// No namespace
// Separating code into namespaces prevents naming conflicts.
// I.e we also use library containing it own version of MyVector class.
// If both of them are in global namespace compiler gets ambigous call in case of using MyVector
class.
template <typename T>
class MyVector : public std::vector<T>
{
public:
    MyVector()
    {
        // Assignment instead of initializer list
        // In this case m_ref is firstly initialized, then size_t pointer with value 1 is assigned to it.
        // Using member initalizer list allows to achieve the same result in one step, which is faster.
        // One more benefit is shorter code.
```

```
        m_ref_ptr = new size_t(1);
        // Assignment instead of initializer list
        m_names = new std::vector<std::string>();
    }

    // Order of initialization
    // Member variables are initialized in order of declaraction, not in order of initialization list.
    // In this case changing doesn't change output of a program, but in some cases it might be misleading.
    // It's good habit so it should be changed anyway.
    MyVector(const MyVector &other)
        : std::vector<T>(other),
          m_ref_ptr(other.m_ref_ptr),
          m_names(other.m_names)
    {
        (*m_ref_ptr)++;
    }
    ~MyVector()
    {
        // Usage of raw pointers
        // This part of code can be ommited by using smart pointers
        if (--*m_ref_ptr == 0)
        {
            delete m_ref_ptr;
            delete m_names;
        }
    }
    void push_back(const T &obj, const std::string &name)
    {
        copy_names();
        std::vector<T>::push_back(obj);
        m_names->push_back(name);
    }
    std::pair<const T &, const std::string &> operator[](int index) const
    {
        // std::out_of_range
        // There is no protection against calling this operator with negative value
        if (index >= std::vector<T>::size())
        {
            // Exception thrown by pointer
            // Throwing a pointer might create memory issues.
            // If program can't allocate memory for pointer it may override previous exception.
            // It may also be lost during stack unwinding.
            throw new std::out_of_range("Index is out of range");
        }
        return std::pair<const T &, const std::string &>(std::vector<T>::operator[](index),
                                                         (*m_names)[index]);
    }
    // Unintuitive operator overloading
    // First overload returns std::pair<T&, const std::string &>
    // Second returns T&
    // Returning the same type for both would be more consistent
```

```
    const T &operator[](const std::string &name) const
    {
        std::vector<std::string>::const_iterator iter = std::find(m_names->begin(), m_names->end(),
                                                                  name);
        if (iter == m_names->end())
        {
            // Exception thrown by pointer
            // Reasoning above, plus this one uses reference variable "name"
            // It may be deleted during stack unwinding, so this usage may be source of error
            throw new std::invalid_argument(name + " is not found in the MyVector");
        }
        return std::vector<T>::operator[](iter - m_names->begin());
    }

private:
    void copy_names()
    {
        if (*m_ref_ptr == 1)
        {
            return;
        }
        size_t *temp_ref_ptr = new size_t(1);
        std::vector<std::string> *temp_names = new std::vector<std::string>(*m_names);
        (*m_ref_ptr)--;
        m_ref_ptr = temp_ref_ptr;
        m_names = temp_names;
    }

private:
    // Use copy-on-write idiom for efficiency (not a premature optimization)

    // Usage of raw pointers instead of smart ones
    // Both m_names and m_ref_ptr are raw pointers, but we can use shared_ptrs for both instead.
    // Smart pointers are less error prone, and they express ownership of a resource.
    // In case of raw pointers usage at first glance we don't know if:
    // - class owns resource
    // - class shares resource
    std::vector<std::string> *m_names;

    // Unprecise variable name
    // Name could be more precise ex. m_ref_cnt, as as it is intended to show current m_names
reference count.
    size_t *m_ref_ptr;
};
#endif // CODEREVIEWTASK_MYVECTOR_HPP
```