Silesian University of Technology

Project Report

Implementation of a custom Vector class

Author: Andrii Bobchuk

Teacher: Dr. Inż. Agnieszka Danek

January 2022

Analysis of the problem

The central goal of this microproject is to implement a custom vector container without using any standard template library containers. This class should act like a normal vector and have a convenient programming interface.

<u>Feature list:</u>

- Implementation of the entire class as a template.
- Custom iterator class.
- Default constructor/Constructor with the initializer list.
- Serialization/Deserialization.
- Search and Sort functions.
- Modifiers (push, pop, emplace).
- Usage of smart pointers.

External Specification

This class has been tested to work with all (or almost all) primitive and struct data types.

Here is how you can declare and initialize a new vector of different types:

```
/// Declaration.
MyVector<float> myFloatVector;
MyVector<double> myDoubleVector;
MyVector<bool> myBoolVector;

/// Initialization.
MyVector<int> myIntVector = {1, 2, 3, 4, 5, 6};
MyVector<std::string> myStringVector = {"1", "2", "3", "4", "5", "6"};
MyVector<char> myCharVector = {'1', '2', '3', '4', '5', '6'};
MyVector<Person> myPeopleVector = {
        Person( newName: "Andriy",  newAge: 19),
        Person( newName: "Viktor",  newAge: 18)
};
```

The following picture demonstrates how to use the ***modification member functions***:

```
myStringVector.pushBack( element: "7");
myPeopleVector.emplaceBack("Youssef", 19);
myStringVector.popBack();
myStringVector.clear();
```

This vector class also has an **emplaceBack()** function with a variable parameter number which allows the user to pass the arguments needed to invoke the constructor of the object he wants to add.

The next example shows the uses of ***accessing vector’s element by its index***:

```
// Element access examples
std::cout << myIntVector[3];
std::cout << myIntVector.at( position: 3);
std::cout << myStringVector[3];
std::cout << myStringVector.at( position: 3);
std::cout << myCharVector[3];
std::cout << myCharVector.at( position: 3);
```

Similarly to the standard class the user has two ways of accessing the element – using the function **at()** or **operator []**

Getting the size and capacity of a vector:

```
std::cout << myIntVector.getSize();
std::cout << myIntVector.getCapacity();
```

In order to search for an element in a whole vector or in a part of it you can use the following functions:

```cpp
std::cout << myStringVector.find( element: "2");
std::cout << myStringVector.find( begin: 2,  end: 6,  element: "4");

// You can also pass custom comparators as arguments:
myPeopleVector.sort( compare: PersonAgeComparator());
myPeopleVector.print();
```

You can also ***sort the elements*** in container in ascending or descending order:

```cpp
// Sort functions:
myIntVector.sort();
myIntVector.sortDescending();
```

There is also a custom iterator class which can be used to go through all elements in a vector.
Here are some examples of how can you ***iterate*** through them:

```cpp
// For each iteration:
for(const auto &element : myIntVector) {
    std::cout << element;
}

// Range based iteration:
for (auto it = myIntVector.begin(); it != myIntVector.end(); it++) {
    std::cout << *it;
}
std::cout<<'\n';
```

Internal Specification

MyVector Class Map

Fields:

shared_ptr<T> mData
Size_t mSize
Size_t mCapacity

Private Methods:

void memAlloc(newCapacity)

Inner Classes:

MyIterator

Fields:

T* mIteratorPointer

Methods:

MyIterator operator--
MyIterator operator--(int)
MyIterator operator++
MyIterator operator++(int)
MyIterator operator->
MyIterator operator*
MyIterator operator==
MyIterator operator!=

Public Methods:

Modifiers:

void pushBack(element)
void push(element, position)
T& emplaceBack(constructor arguments)
void popBack()
void clear()

Element access:

T &operator[](position)
T &at(position)

Capacity:

size_t size()
size_t capacity()

Serialization:

void serialize()
void deserialize()

Find/Sort:

int find(element)
int find (begin, end, element)
void sort()
void sortDescending()

This container has all of the member functions as the standard vector class, only that I used another ***naming convention***:

- Class names - nouns in title-case with the first letter of each separate word capitalized
- Methods - verbs in camel case notation.
- Variables - camel case. Member variables are notated as e.g. “mVariable”

When the user calls the constructor of MyVector holding elements of any type, the ***constructor*** is called:

```
MyVector() { memAlloc( requiredCapacity: 0); }
```

Alled: As it is visible, constructor calls the memory allocation function which initially reserves a new block of memory of capacity 0 like a standard vector class.

Here is how this function operates:

```
void memAlloc(size_t requiredCapacity) {

    /// Allocate a new block of memory with a new capacity:
    std::shared_ptr<T> newMemBlock(new T[requiredCapacity]);

    /// Move elements from old block to a new one:
    for (size_t i = 0; i < mSize; i++) {
        newMemBlock.get()[i] = std::move(mData.get()[i]);
    }

    /// Reset mData field to store our new block of memory:
    mData = newMemBlock;

    /// Set a new capacity value
    mCapacity = requiredCapacity;

    std::cout << requiredCapacity << " memory cells allocated\n";
}
```

The reason I decided to make **memAlloc()** a separate function is because although initially we allocate a block of capacity 0, we constantly need to reallocate memory as we push new elements to our container:

```
void pushBack(const T &element) {

    // When not enough capacity we double it, but we cannot double zero
    if (mCapacity == 0) {
        memAlloc( requiredCapacity: 1);
    } else if (mCapacity <= mSize) {
        memAlloc( requiredCapacity: mCapacity * 2);
    }
    mData.get()[mSize] = element;
    mSize++;
};
```

Here in the **pushBack()** function we firstly check if the old capacity was zero. In that case we cannot just double the capacity, we call memAlloc(1). In all other cases - like we are supposed to.

emplaceBack() in its core works the same way:

```
template<typename... Args>
T &emplaceBack(Args &&... args) {

    /*
     * Exactly the same code as push fun, only last line differs
     */

    // When not enough capacity we double it, but we cannot double zero
    if (mCapacity == 0) {
        memAlloc( requiredCapacity: 1);
    } else if (mCapacity <= mSize) {
        memAlloc( requiredCapacity: mCapacity * 2);
    }

    // Instead of making mData[mSize] equal to object we forward all our
    // arguments to the constructor:
    mData.get()[mSize] = T(std::forward<Args>(args)...);;
    mSize++;
    return mData.get()[mSize];
};
```

This was the hardest part of the project.

Now, here is how I implemented ***element access***:

```
// Works for mutable objects:
T &operator[](size_t position) { return mData.get()[position]; }

// Works for const objects:
const T &operator[](size_t position) const { return mData.get()[position]; }
```

Sort function works relying on a bubble sort algorithm. It can sort elements in ascending or descending order and compare elements of all most-common primitive data types. Also provided with the special comparator, it can sort custom structures by their properties.

Iterators. I have implemented the MyIterator as an inner class inside of MyVector.

I wanted to the user to be able to iterate the vector in range-based and for loops, so I had to implement at least begin(), end(), increment, decrement and comparator operations:

```
class MyIterator {
    T *mIteratorPointer; // Pointer to current position of our iterator
public:
    explicit MyIterator(T *ptr) { mIteratorPointer = ptr;}

    // Pre increment:
    MyIterator &operator++() noexcept {
        mIteratorPointer++;
        return *this;
    }
    // Post increment:
    MyIterator operator++(int) {
        MyIterator iterator = *this;
        ++(*this);
        return iterator;
    }

    T *operator->() { return mIteratorPointer; }

    T &operator*() { return *mIteratorPointer; }

    bool operator==(const MyIterator &iteratorToCompareWith) const {
        return mIteratorPointer == iteratorToCompareWith.mIteratorPointer;
    }

    bool operator!=(const MyIterator &iteratorToCompareWith) const {
        return mIteratorPointer != iteratorToCompareWith.mIteratorPointer;
    }
};
```

Tests

I have tested the various modules of the program and many problems were mostly ideological, so I could have named the bugs as features but I was trying to implement everything as close to the standard container class as possible.

Some of the problematic questions were:

- **In MyVector constructor.** I could've just made an initial capacity as one or two. No one declares vectors for no reason so it makes sense to reserve some space in advance. I would also reduce some boilerplates in code related to that. However the std vector's initial capacity is 0, so I made it 0;

- **Serialization.** In the latest version of this function the algorithm writes the content of a vector to a stringstream and then directly to a binary file. It works with vectors containing objects of all primitive types. Also, it works with a simple structure Person I have provided for tests in MyVector.h.

```cpp
void serialize(const std::string &fileName) {

    // Opening the binary file by the name:
    std::ofstream outFileStream(fileName, std::ios::binary);

    if (outFileStream.good()) {

        // Firstly, write the number of elements in our vector:
        outFileStream.write((char *) &mSize, sizeof(mSize));

        for (size_t i = 0; i < mSize; i++) {

            std::stringstream stringStream; // stringStream representing each element
            stringStream << mData.get()[i];
            size_t stringSize = stringStream.str().size();

            // Writing the length of generated string:
            outFileStream.write((char *) &(stringSize), sizeof(stringSize));

            // Writing this string:
            outFileStream.write((char *) (stringStream.str().data()), sizeof(char) * stringSize);
        }
        outFileStream.close();
    }
}
```

I was fixing many minor and major bugs, but could have missed something.

So, for the test purposes I provided a separate file with a convenient (as I think) way to test all the key modules of this class. You can just enable the block you are currently testing so that there is no mess in console:

```

    /* =============================================================
     *                                    ELEMENT ACCESS
     * =============================================================
#if 0
    ...
#endif

    /* =============================================================
     *                                    CAPACITY
     * =============================================================
#if 0
    ...
#endif

    /* =============================================================
     *                                    SERIALIZATION
     * =============================================================
#if 0
    ...
#endif
```

I have also provided a member print function not to write loops every time:

```
void print() {
    try {
        for (size_t i = 0; i < mSize; i++) {
            std::cout << mData.get()[i] << "; ";
        }
        std::cout << std::endl;
    } catch (std::exception e) {
        std::cout << "Ooops " << e.what() << std::endl;
    }
}
```