```
#ifndef __MYVECTOR_H__
#define __MYVECTOR_H__

#include <utility>
#include <vector>

template <typename DataType>
class MyVector
{
  protected:
    /* data */
    size_t theSize;                         // the number of data elements the
vector is currently holding
    size_t theCapacity;                     // maximum data elements the vector can
hold
    DataType *data;                         // address of the data storage

  public:

    static const size_t SPARE_CAPACITY = 16;   // initial capacity of the vector

    // default constructor
    explicit MyVector(size_t initSize = 0) :
        theSize{initSize},
        theCapacity{initSize + SPARE_CAPACITY}
    {
        // code begins
        data = new DataType[theCapacity];
        // code ends
    }

    // copy constructor
    MyVector(const MyVector & rhs) :
        theSize{rhs.theSize},
        theCapacity{rhs.theCapacity}
    {
        // code begins
        data = new DataType[theCapacity];
        for (int i = 0; i < theSize; i++)
        {
            data[i] = rhs.data[i];
        }
        // code ends
    }

    // move constructor
    MyVector(MyVector&& rhs):
        theSize{rhs.theSize},
        theCapacity{rhs.theCapacity},
        data{rhs.data}
    {
        // code begins
        rhs.data = nullptr;
        rhs.theSize = 0;
        rhs.theCapacity = 0;
        // code ends
    }

    // copy constructor from STL vector implementation
```

```
MyVector(const std::vector<DataType> & rhs) :
    theSize{rhs.size()},
    theCapacity{rhs.size() + SPARE_CAPACITY}
{
    // code begins
    data = new DataType[theCapacity];
    for (int i = 0; i < theSize; i++)
    {
        data[i] = rhs[i];
    }
    // code ends
}

// destructor
~MyVector(){
    // code begins
    delete [] data;
    // code ends
};

// copy assignment
MyVector & operator= (const MyVector& rhs)
{
    // code begins
    MyVector copy = rhs;
    std::swap(*this, copy);
    return *this;
    // code ends
}

// move assignment
MyVector & operator= (MyVector && rhs)
{
    // code begins
    std::swap(theSize, rhs.theSize);
    std::swap(theCapacity, rhs.theCapacity);
    std::swap(data, rhs.data);

    return *this;
    // code ends
}

// change the size of the array
void resize(size_t newSize)
{
    // code begins
    if (newSize > theCapacity)
    {
        reserve(newSize * 2);
    }

    theSize = newSize;
    // code ends
}

// allocate more memory for the array
void reserve(size_t newCapacity)
{
    // code begins
```

```
    theCapacity = newCapacity;
    DataType *newArray = new DataType[newCapacity];
    for (int i = 0; i < theSize; i++)
    {
        newArray[i] = std::move(data[i]);
    }

    std::swap(data, newArray);
    delete [] newArray;
    // code ends
}

// data access operator (without bound checking)
DataType & operator[] (size_t index)
{
    // code begins
    return data[index];
    // code ends
}

const DataType & operator[](size_t index) const
{
    // code begins
    return data[index];
    // code ends
}

// check if the vector is empty; return TURE if the vector is empty
bool empty() const
{
    // code begins
    return size() == 0;
    // code ends
}

// returns the size of the vector
size_t size() const
{
    // code begins
    return theSize;
    // code ends
}

// returns the capacity of the vector
size_t capacity() const
{
    // code begins
    return theCapacity;
    // code ends
}

// insert an data element to the end of the vector
void push_back(const DataType & x)
{
    // code begins
    if (theSize == theCapacity)
    {
        reserve(2 * theCapacity + 1);
```

```
    }

    data[theSize++] = std::move(x);
    // code ends
}

void push_back(DataType && x)
{
    // code begins
    if (theSize == theCapacity)
    {
        reserve(2 * theCapacity + 1);

    }

    data[theSize++] = std::move(x);
    // code ends
}

// append a vector as indicated by the parameter to the current vector
MyVector<DataType>& append(MyVector<DataType> && rhs)
{
    // code begins
    for (int i = 0; i < rhs.theSize; i++)
    {
        push_back(rhs.data[i]);
    }

    return *this;
    // code ends
}

// remove the last data element from the array
void pop_back()
{
    // code begins
    --theSize;
    // code ends
}

// returns the last data elemtn from the array
const DataType& back() const
{
    // code begins
    return data[theSize - 1];
    // code ends
}

// iterator implementation

typedef DataType* iterator;
typedef const DataType* const_iterator;

iterator begin()
{
    // code begins
    return &data[0];
    // code ends
}
```

```
    const_iterator begin() const
    {
        // code begins
        return &data[0];
        // code ends
    }

    iterator end()
    {
        // code begins
        return &data[size()];
        // code ends
    }

    const_iterator end() const
    {
        // code begins
        return &data[size()];
        // code ends
    }

};


#endif //__MYVECTOR_H__
```