1. Running Environment

Visual Studio 2019에서 Debug x86으로 진행

2. Code Snippets

```
#include <iostream>
#include <algorithm>
#include <cstdlib>

class MyDoubleVector {
    typedef double value_type;
private:
    // Member data;
    static const size_t Default_Capacity = 100;
    double* data;
    size_t capacity;
    size_t size;

public:
    // Default function
    MyDoubleVector(size_t init_capacity = Default_Capacity);    // 생성자--
    MyDoubleVector(const MyDoubleVector& v);                    // 복사 생성자--
    ~MyDoubleVector();                                          // 파괴자

    // Operator Overloading
    MyDoubleVector& operator=(const MyDoubleVector& v);         // =: 깊은 복사--
    void operator+=(const MyDoubleVector& v);                   // +=: 벡터에 매개변수 벡터 원소를 추가--
    double operator[] (size_t idx);                             // []: 요청된 벡터 인덱스값 출력--
    MyDoubleVector operator+(const MyDoubleVector& v);          // +: 벡터합--
    MyDoubleVector operator-(const MyDoubleVector& v);          // -: 벡터차--
    double operator*(const MyDoubleVector& v);                  // *: 스칼라곱--
    MyDoubleVector& operator-();                                // -: 원소를 전부 부호바꿈--
    bool operator==(const MyDoubleVector& v);                   // ==: 2개의 벡터의 동일 유무--
    MyDoubleVector& operator()(value_type n);                   // (): 벡터의 원소를 전부 매개변수로 바꿈

    // Member Function
    void pop_back();                                            // 마지막 원소 제거--
    void push_back(value_type x);                               // 마지막에 원소 추가--
    size_t Capacity() const;                                    // 벡터에 할당된 저장공간 반환--
    size_t Size() const;                                        // 벡터 원소의 개수 반환--
    void reserve(size_t n);                                     // 최소 n개의 원소를 가질 수 있는 저장공간 요청--
    bool empty() const;                                         // 벡터가 비었는지 유무 출력--
    void clear();                                               // 모든 벡터 원소를 제거--
};
```

클래스 선언부 헤더파일은 다음과 같고, 각 멤버 함수의 역할은 주석에 써있는 바와 같다.

```
MyDoubleVector::MyDoubleVector(size_t init_capacity)
// Postcondition: dynamically allocate memory of size 'init_capacity', and initial size = 0
{
    data = new value_type[init_capacity];
    capacity = init_capacity;
    size = 0;
}

MyDoubleVector::MyDoubleVector(const MyDoubleVector& v)
// Postcondition: dynamically allocate memory new memory and perform deep copy
{
    data = new value_type[v.Capacity()];
    capacity = v.Capacity();
    size = v.size;
    std::copy(v.data, v.data + capacity, data);
}

MyDoubleVector::~MyDoubleVector()
// Precondition: object should exist
// Postcondition: free the memory of the object
{
    delete [] data;
}
```

생성자와 복사생성자, 파괴자는 위와 같다. 복사 생성자의 경우, 참조로 입력받은 벡터 v.capacity만큼의 메모리를 동적할당받고, 그 메모리에 v의 데이터를 복사해준다. (깊은 복사)

```cpp
MyDoubleVector& MyDoubleVector::operator=(const MyDoubleVector& v)
// Precondition: allocated memory should exist
// Postcondition: conduct deep copy
{
    if (this == &v) return *this;
    if (capacity != v.capacity)
    {
        delete[] data;
        data = new value_type[v.capacity];
        capacity = v.capacity;
    }
    size = v.size;
    std::copy(v.data, v.data + capacity, data);
    return *this;
}

void MyDoubleVector::operator+=(const MyDoubleVector& v)
// Postcondition: add all the elements of parameter vector to object vector
{
    if (size + v.size > capacity) reserve(size + v.size);
    std::copy(v.data, v.data + v.size, data + size);
    size += v.size;
}

double MyDoubleVector::operator[] (size_t idx)
// Precondition: idx < size
// Postcondition: return idx_th element
{
    assert(idx < size);
    return data[idx];
}
```

=, +=, []연산자를 각각 오버로딩 하였다. =연산자의 경우, 깊은 복사를 하는 것은 맞지만, 복사생성자와는 다르게 기존에 할당받아 존재하는 메모리에 v.data를 복사해준다. 만약 capacity의 크기가 서로 다르다면, 해당 객체의 메모리를 해제 후, v.capacity만큼의 메모리를 다시 할당받아 데이터를 복사한다.

```
MyDoubleVector MyDoubleVector::operator+(const MyDoubleVector& v)
// Precondition: size == v.size
// Postcondition: return the vector sum of 2 operand vectors
{
    assert(size == v.size);
    MyDoubleVector sum(v.capacity);
    sum.size = size;
    for (size_t i = 0; i < v.size; i++)
    {
        sum.data[i] = data[i] + v.data[i];
    }
    return sum;
}
```

+ 연산자의 경우, 우선 v와 this 객체의 사이즈가 같은지 확인한다. this 객체와 v객체의 같은 번째 데이터를 더한 후 새로 선언해준 MyDoubleVector 객체 sum의 인덱스에 그 값을 넣어준다. -와 *(반환형 double)연산자도 비슷하게 작성하였다.

```
bool MyDoubleVector::operator==(const MyDoubleVector& v)
// Precondition: (*data == *v.data && size == v.size)
// Postcondition: returns wether or not 2 vectors are same
{
    if (*data == *v.data && size == v.size)
    {
        std::cout << "same!" << std::endl;

        return true;
    }
    else
    {
        std::cout << "not same!" << std::endl;
        return false;
    }
}
```

==연산자는 this 객체와 v 객체의 데이터가 같은지 판단 후, boolean으로 그 여부를 리턴한다. 시각화를 위해 같음의 유무도 출력해주었다.

```
void MyDoubleVector::pop_back()
// Precondition: size >= 1
// Postcondition: eliminate the last element of the object vector
{
    assert(size >= 1);
    data[--size] = NULL;
}

void MyDoubleVector::push_back(value_type x)
// Postcondition: insert x to the last of the object vector
{
    if(size < capacity) data[size++] = x;
    else
    {
        reserve(size + 1);
        data[size++] = x;
    }
}
```

push_back, pop_back 구현 부분은 위와 같다. push_back의 경우, capacity가 더 필요할 경우, size+1만큼 reserve 해주고, 데이터의 인덱스를 증가시키며 x를 넣어준다.

```
void MyDoubleVector::reserve(size_t new_capacity)
// Precondition: new_capacity >= 0
// Postcondition: request the memory capacity to hold n elements
{
    assert(new_capacity >= 0);
    value_type *larger_vector;
    if(new_capacity == capacity) return;
    if(new_capacity < size) new_capacity = size;

    larger_vector = new value_type[new_capacity];
    std::copy(data, data + size, larger_vector);
    delete[] data;
    data = larger_vector;
    capacity = new_capacity;
}
```

reserve는 할당하고자 하는 새로운 사이즈를 입력받고, double형의 larger_vector로 입력받은 사이즈 만큼의 공간을 동적할당 받는다. 그 후, 기존의 데이터를 larger_vector에 복사해주고 삭제하고, data를 larger_vector로 초기화해준다.

```cpp
#include <iostream>
#include "MyDoubleVector.h"
#include <algorithm>

using namespace std;

int main()
{
    // Size(), Capacity(), operator[]
    // constructor
    MyDoubleVector a;
    cout << "examine constructor" << endl;
    cout << "a.capacity: " << a.Capacity() << "₩t₩t" << "a.size: " << a.Size() << endl;
    cout << endl;

    // push_back
    a.push_back(1);
    a.push_back(3.14);
    cout << "examine push_back" << endl;
    cout << "a.capacity: " << a.Capacity() << "₩t₩t" << "a.size: " << a.Size() << endl;
    for (size_t i = 0; i < a.Size(); i++)
    {
        cout << "a[" << i << "]: " << a[i] << "₩t₩t";
    }
    cout << "₩n" << endl;
```

위에서 구현한 클래스의 각 멤버함수는 위 사진과 같은 방식으로 테스트해 주었다.

실행 결과는 다음과 같다.

```
Microsoft Visual Studio 디버그 콘솔
examine constructor
a.capacity: 100         a.size: 0

examine push_back
a.capacity: 100         a.size: 2
a[0]: 1         a[1]: 3.14

examine copy constructor
b.capacity: 100         b.size: 2
b[0]: 1         b[1]: 3.14

examine pop_back
b.capacity: 100         b.size: 1
b[0]: 1

examine operator=
c.capacity: 100         c.size: 2
c[0]: 1         c[1]: 3.14

examine operator+=
a.capacity: 100         a.size: 4
a[0]: 1         a[1]: 3.14              a[2]: 1         a[3]: 3.14

examine operator+
sum[0]: 2               sum[1]: 8.44

examine operator-
dif[0]: 0               dif[1]: 2.16

examine operator*
dot product value: 17.642
examine operator-
a[0]: -1                a[1]: -3.14             a[2]: -1                a[3]: -3.14

examine operator==
not same!

examine operator()
a[0]: 5         a[1]: 5         a[2]: 5         a[3]: 5

v1.Capacity(): 100
examine reserve
v1.Capacity(): 105

examine empty
Vector is empty!
Vector is not empty!

examine clear
a.Size(): 0
```

3. Discussion(고찰)

코드를 작성하면서 클래스의 기본 구조와, .h와 .cpp가 어떤 기준으로 나뉘는지에 대해 알 수 있었고, 오류를 찾아 나아가는 과정에서 몇가지를 알게 되었고, 이는 다음과 같다.

1. 생성자 선언부에 Default argument가 있고, main에서 이를 사용하고 싶을 경우, 생성자 옆에 매개변수 괄호 () 없이 바로 ;를 기입해야한다.
2. 생성자 선언부에서 Default argument를 넣어주었을 때, 이를 실제로 구현하는 부분에서는 Default argument를 써주지 않는다.
3. 클래스 멤버함수 내에서도 그 클래스타입의 인지를 생성 및 리턴이 가능하다.