KWANGWOON UNIVERSITY

자 료 구 조

- Homework #1 -

(MyDoubleVector)

* Visual Studio 2015 environment

담당 교수님 : 김용혁 교수님
제 출 날 짜 : 2017. 10. 13.
학 과 : 컴퓨터소프트웨어학과
학 번 : 2014707040
이 름 : 유 진 혁

KWANGWOON UNIVERSITY

목 차

1. 클래스 멤버 구성

(1). public 영역

```
public:
    static const int INIT_CAPACITY = 1;
    static const int CAPACITY_UP = 3;
    static const int ERR_OUT_RANGE = 1;
    static const int ERR_SIZES = 2;
    MyDoubleVector();
    MyDoubleVector(const MyDoubleVector& v);
    ~MyDoubleVector();
    MyDoubleVector& operator=(const MyDoubleVector& source);
    void operator+=(const MyDoubleVector& addend);
    double& operator[](size_t n);
    const MyDoubleVector operator+(const MyDoubleVector& v);
    const MyDoubleVector operator-(const MyDoubleVector& v);
    double operator*(const MyDoubleVector& v);
    const MyDoubleVector operator-();
    bool operator==(const MyDoubleVector& v);
    void operator()(double value);
    void pop_back();
    void push_back(double x);
    size_t capacity() const;
    size_t size() const;
    void reserve(size_t n);
    bool empty() const;
    void clear();
```

<그림 1. MyDoubleVector 클래스의 public 영역>

과제에서 제시한 모든 함수들을 클래스의 멤버 함수로 구현하여 public 영역에 넣어주었다. public 영역에 함수들 이외에도 추가로 몇 가지 상수들을 정의하여 넣어주었다.

INIT_CAPACITY는 MyDoubleVector 클래스의 객체 생성 시 capacity의 초기 값이다. CAPACITY_UP은 vector container의 size가 그의 capacity만큼 다 찼을 때, push_back 함수로 값을 넣어줄 경우 추가로 증가시킬 capacity의 크기이다. ERR_OUT_RANGE는 할당된 메모리를 넘어 요소를 참조하려고 할 때 발생하는 에러 코드 값이고, ERR_SIZES는 이항연산자 +, -, * 연산 시 두 피연산자 객체의 size가 다를 때 발생하는 에러 코드 값이다.

(2). private 영역

```
private:
    double *data;
    size_t used;
    size_t space;
```

<그림 2. MyDoubleVector 클래스의 private 영역>

vector 구성에 필요한 변수들을 private 영역에 넣어주었다. data는 동적 할당된 메모리 공간을 가리킬 포인터 변수이다. used는 현재 동적 배열에 값이 들어있는 요소의 길이(size)이고, space는 할당받은 메모리 공간의 길이(capacity)를 나타낸다.

2. 각 함수 별 소스코드 설명

(1). MyDoubleVector()

```
MyDoubleVector::MyDoubleVector()
{
    // precondition: When objects are created.
    // postcondition: The memory is allocated as much as INIT_CAPACITY, member variables are initialized.
    data = new double[INIT_CAPACITY];
    space = INIT_CAPACITY;
    used = 0;
}
```

<그림 3. MyDobleVector 클래스의 Default constructor>

MyDoubleVector 클래스의 Default constructor이다. MyDoubleVector 클래스의 객체 생성 시, 멤버변수들을 초기 값으로 초기화한다. INIT_CAPACITY 상수 크기만큼의 double 값을 담는 메모리 공간을 생성하고 data 포인터 변수가 그를 가리키게 한다. INIT_CAPACITY의 값은 1로 설정하였다. space 변수엔 capacity 초기 값, 즉 INIT_CAPACITY의 값이 들어가고, 배열이 비어있으므로 used는 0으로 초기화된다.

(2). MyDoubleVector(const MyDoubleVector& v)

```
MyDoubleVector::MyDoubleVector(const MyDoubleVector& v)
{
    // precondition: When an object of the class is returned by value,
    //               When an object of the class is passed (to a function) by value as an argument,
    //               When an object is constructed based on another object of the same class.
    // postcondition: The object is copied.
    data = new double[v.space];
    space = v.space;
    used = v.used;
    copy(v.data, v.data + used, data);
}
```

<그림 4. MyDoubleVector 클래스의 Copy constructor>

MyDoubleVector 클래스의 Copy constructor이다. 기존 객체를 복사하면서 새 인스턴스를 만든다. 기존 객체의 space 크기만큼 double형 동적 배열을 할당하고, space와 used 변수를 복사한다. 새롭게 할당된 배열에 기존 요소들을 Deep copy해주기 위해 copy 함수를 이용했다. 기존 객체 v의 data 시작위치(v.data)부터 used 크기만큼(v.data + used)의 요소를 새 data가 가리키는 곳으로 복사한다.

(3). ~MyDoubleVector()

```
MyDoubleVector::~MyDoubleVector()
{
    // precondition: When objects are destroyed (deallocated).
    // postcondition: Frees the allocated memory.
    delete[] data;
}
```

<그림 5. MyDoubleVector 클래스의 Destructor>

MyDoubleVector 클래스의 Destructor이다. data가 가리키던 동적 할당된 메모리를 해제한다.

2. 각 함수 별 소스코드 설명

(4). MyDoubleVector& operator=(const MyDoubleVector& source)

```
MyDoubleVector& MyDoubleVector::operator=(const MyDoubleVector& source)
{
    // postcondition: RHS object is assigned to LHS object.
    double *new_data;
    if (this == &source)
        return *this;
    if (space != source.space)
    {
        new_data = new double[source.space];
        delete[] data;
        data = new_data;
        space = source.space;
    }
    used = source.used;
    copy(source.data, source.data + used, data);
    return *this;
}
```

<그림 6. MyDoubleVector 클래스의 operator= 오버로딩>

Chaining assignment가 가능하도록 객체의 레퍼런스를 반환한다. 대입연산자의 두 피연산자가 같을 경우 자기 자신을 반환하도록 조건 분기를 두었다. 두 피연산자의 capacity, 즉 space의 크기가 다르다면 새로 선언한 new_data 포인터 변수에 RHS 객체의 space 크기만큼의 동적 배열을 할당하고 기존 data(LHS 객체의 data)의 메모리를 해제한 뒤 새로 할당한 배열을 가리키도록 대입 연산을 한다. 두 피연산자의 변수 space의 값도 같게 만든다. 두 피연산자의 space 크기가 같다면 앞의 과정은 생략하게 된다. 그러고 나서 두 피연산자의 변수 used를 같게 만들어주고 copy 함수를 이용하여 각각의 요소들을 Deep copy한다. 마지막으로 이렇게 수정된 자기 자신(LHS)을 반환한다.

(5). void operator+=(const MyDoubleVector& addend)

```
void MyDoubleVector::operator+=(const MyDoubleVector& addend)
{
    // postcondition: Appends RHS object to LHS one.
    if (used + addend.used > space)
        reserve(used + addend.used);
    copy(addend.data, addend.data + addend.used, data + used);
    used += addend.used;
}
```

<그림 7. MyDoubleVector 클래스의 operator+= 오버로딩>

RHS 객체의 요소를 LHS 객체(자기 자신 this)의 요소들 끝에 덧붙이는 연산자이다. LHS의 capacity가 덧붙여질 만큼 충분하지 않다면 reserve 함수로 메모리 공간을 늘린다. reserve 함수에 대해서는 뒤에서 설명하도록 하겠다. copy 함수로 RHS 객체(addend)의 첫 요소부터 마지막 요소까지를 LHS 객체의 요소 끝에 복사하고 LHS 객체의 used 값을 RHS 객체의 used만큼 더해 저장한다.

2. 각 함수 별 소스코드 설명

(6). double& operator[](size_t n)

```
double& MyDoubleVector::operator[](size_t n)
{
    // precondition: n < used
    // postcondition: Returns a reference to the element at the requested position in the vector container.
    if (n >= used)
    {
        cout << "The requested position is out of range." << endl;
        exit(ERR_OUT_RANGE);
    }
    return *(data + n);
}
```

<그림 8. MyDoubleVector 클래스의 operator[] 오버로딩>

인자로 받은 위치의 요소 값 레퍼런스를 반환하는 연산자이다. 연산의 반환 값에 대입을 받고 수정도 가능해야하므로 레퍼런스를 반환한다. 인자로 받은 n의 값이 used보다 클 때, 즉 값이 저장되어 있는 메모리 범위를 벗어날 때 에러 메시지를 출력하며 프로그램이 종료된다. 그렇지 않을 경우, 객체 요소의 시작위치로부터 n을 더한 위치가 dereference 연산되어 반환된다.

(7). const MyDoubleVector operator+(const MyDoubleVector& v)

```
const MyDoubleVector MyDoubleVector::operator+(const MyDoubleVector& v)
{
    // precondition: When the sizes of the two operands is the same.
    // postcondition: Returns an object that is a vector-sum of the two operand objects.
    MyDoubleVector answer;
    if (this->used != v.used)
    {
        cout << "The sizes of the two operands is not the same." << endl;
        exit(ERR_SIZES);
    }
    answer.reserve(used);
    for (size_t i = 0; i < used; i++)
    {
        answer.data[i] = data[i] + v.data[i];
        answer.used++;
    }
    return answer;
}
```

<그림 9. MyDoubleVector 클래스의 operator+ 오버로딩>

두 피연산자의 요소끼리의 합이 담긴 객체를 반환하는 연산자이다. 요소끼리의 합을 구하기 위해선 두 피연산자 객체의 size가 같아야한다. 같지 않을 경우 에러 메시지를 출력하며 프로그램이 종료된다. 함수 내에서 answer라는 MyDoubleVector 클래스의 객체를 하나 만든다. answer 객체 내의 메모리 공간을 피연산자의 used만큼 reserve 함수로 늘린다. 반복문을 통해 두 피연산자의 요소끼리 더한 값을 answer의 data 동적 배열에 하나씩 채운다. 채움과 동시에 answer의 used 변수 값도 증가시킨다. 이렇게 만들어진 객체 answer를 반환하고 반환된 answer의 값들이 수정되어선 안 되므로 const로 반환한다.

2. 각 함수 별 소스코드 설명

(8). const MyDoubledVector operator-(const MyDoubleVector& v)

```
const MyDoubleVector MyDoubleVector::operator-(const MyDoubleVector& v)
{
    // precondition: When the sizes of the two operands is the same.
    // postcondition: Returns an object that is a vector-difference of the two operand objects.
    MyDoubleVector answer;
    if (this->used != v.used)
    {
        cout << "The sizes of the two operands is not the same." << endl;
        exit(ERR_SIZES);
    }
    answer.reserve(used);
    for (size_t i = 0; i < used; i++)
    {
        answer.data[i] = data[i] - v.data[i];
        answer.used++;
    }
    return answer;
}
```

<그림 10. MyDoubleVector 클래스의 Binary operator- 오버로딩>

두 피연산자의 요소끼리의 차가 담긴 객체를 반환하는 연산자이다. 요소끼리의 차를 구하기 위해선 두 피연산자 객체의 size가 같아야한다. 같지 않을 경우 에러 메시지를 출력하며 프로그램이 종료된다. 함수 내에서 answer라는 MyDoubleVector 클래스의 객체를 하나 만든다. answer 객체 내의 메모리 공간을 피연산자의 used만큼 reserve 함수로 늘린다. 반복문을 통해 두 피연산자의 요소끼리 뺀 값을 answer의 data 동적 배열에 하나씩 채운다. 채움과 동시에 answer의 used 변수 값도 증가시킨다. 이렇게 만들어진 객체 answer를 반환하고 반환된 answer의 값들이 수정되어선 안 되므로 const로 반환한다.

(9). double operator*(const MyDoubleVector& v)

```
double MyDoubleVector::operator*(const MyDoubleVector& v)
{
    // precondition: When the sizes of the two operands is the same.
    // postcondition: Returns the scalar product value of the two operand objects.
    double scalar = 0.0;
    if (this->used != v.used)
    {
        cout << "The sizes of the two operands is not the same." << endl;
        exit(ERR_SIZES);
    }
    for (size_t i = 0; i < used; i++)
        scalar += data[i] * v.data[i];
    return scalar;
}
```

<그림 11. MyDoubleVector 클래스의 operator* 오버로딩>

두 피연산자의 요소끼리의 내적을 구해 반환하는 연산자이다. 요소끼리의 내적을 구하므로 두 피연산자 객체의 size가 같아야한다. 같지 않을 경우 에러 메시지를 출력하며 프로그램이 종료된다. double형 변수 scalar를 선언하여 0으로 초기화한 후 반복문을 통해 요소끼리의 곱을 더해가며 저장한다. 끝 요소끼리 곱까지 scalar에 더한 후 scalar를 반환한다.

2. 각 함수 별 소스코드 설명

(10). const MyDoubleVector operator-()

```
const MyDoubleVector MyDoubleVector::operator-()
{
    // postcondition: Returns an object of which each element is the unary negation.
    MyDoubleVector answer;
    answer.reserve(used);
    for (size_t i = 0; i < used; i++)
    {
        answer.data[i] = -data[i];
        answer.used++;
    }
    return answer;
}
```

<그림 12. MyDoubleVector 클래스의 Unary operator- 오버로딩>

객체의 요소 값들을 음수화 또는 양수화(Negation)하는 연산자이다. answer라는 객체를 생성하고 호출한 객체의 size만큼 메모리 공간을 늘린다. 호출한 객체의 요소 값들을 negation한 후 answer의 data가 가리키는 배열 공간에 채운다. 동시에 answer의 used 값도 증가시킨다. 이를 요소의 끝에 도달할 때까지 반복한 후 answer 객체를 반환한다. 반환된 answer의 값들은 수정되어선 안 되므로 const로 반환한다.

(11). bool operator==(const MyDoubleVector& v)

```
bool MyDoubleVector::operator==(const MyDoubleVector& v)
{
    // postcondition: Returns true if sizes of two operand vectors are same and thier each element is the same.
    //                Otherwise returns false.
    size_t check = 0;
    if (used == v.used)
    {
        for (size_t i = 0; i < used; i++)
        {
            if (data[i] == v.data[i])
                check++;
        }
        return check == used;
    }
    return false;
}
```

<그림 13. MyDoubleVector 클래스의 operator== 오버로딩>

두 객체의 요소를 비교하여 길이와 내용이 모두 같다면 true, 하나라도 다르다면 false를 반환하는 연산자이다. 먼저 피연산자의 used를 비교하고 각 요소의 값을 하나하나 비교하여 같으면 check 변수를 1 증가시킨다. check 변수의 값이 used와 같으면 모든 요소의 값이 같다는 것이므로 true를 반환한다. 그렇지 않으면 false를 반환한다.

2. 각 함수 별 소스코드 설명

(12). void operator()(double value)

```
void MyDoubleVector::operator()(double value)
{
    // precondition: When the vector container is not empty.
    // postcondition: Makes every element of this object be the value of the operand.
    for (size_t i = 0; i < used; i++)
        data[i] = value;
}
```

<그림 14. MyDoubleVector 클래스의 operator() 오버로딩>

객체에 저장된 요소 값들을 모두 인자로 받은 value로 바꾸는 연산자이다. 동적 배열에 값이 저장되어 있는 요소의 길이만큼, 즉 size(used)만큼 value로 채워진다. 때문에 used가 0이면 ()연산을 해도 아무런 변화가 일어나지 않는다. 반복문을 통해 각 요소에 접근해 값을 바꾼다.

(13). void pop_back()

```
void MyDoubleVector::pop_back()
{
    // precondition: When the vector container is not empty.
    // postcondition: Removes the last element in the vector.
    if (!empty())
        reserve(--used);
}
```

<그림 15. MyDoubleVector 클래스의 pop_back 함수>

객체의 마지막 요소를 제거하는 함수이다. 요소를 제거하기 위해선 배열이 비어있지 않아야 하므로 empty 함수로 여부를 판단한다. empty 함수에 대해선 뒤에서 설명하겠다. 배열이 비어있지 않으면 used를 하나 줄이고 줄어든 used만큼 capacity도 줄이기 위해 reserve 함수를 호출한다. 이렇게 하면 마지막 요소가 사라지고 해당 위치를 참조할 수도 없게 된다.

(14). void push_back(double x)

```
void MyDoubleVector::push_back(double x)
{
    // precondition: When the memory is not full.
    // postcondition: Adds a new element at the end of the vector.
    if (used == space)
        reserve(used + CAPACITY_UP);
    data[used++] = x;
}
```

<그림 16. MyDoubleVector 클래스의 push_back 함수>

객체의 요소 끝에 인자로 받은 x 값을 새 요소로 추가하는 함수이다. capacity가 다 차서 할당받은 메모리 공간이 부족할 경우 reserve 함수로 공간을 늘리고 새 요소를 추가한다. 이 때, capacity는 상수 CAPACITY_UP만큼 증가한다. 요소를 추가한 뒤 used의 값을 1 증가시킨다.

2. 각 함수 별 소스코드 설명

(15). size_t capacity() const

```
size_t MyDoubleVector::capacity() const
{
    // postcondition: Returns the size of the allocated storage space for the elements of the vector container.
    return space;
}
```

<그림 17. MyDoubleVector 클래스의 capacity 함수>

객체의 private 영역에 선언된 변수 space를 반환하는 getter 함수이다. 즉, 동적으로 할당받은 배열 공간의 크기를 반환한다. 함수 내에서 변수의 값을 바꾸면 안 되므로 함수 선언문 끝에 const를 붙여주었다.

(16). size_t size() const

```
size_t MyDoubleVector::size() const
{
    // postcondition: Returns the number of elements in the vector container.
    return used;
}
```

<그림 18, MyDoubleVector 클래스의 size 함수>

객체의 private 영역에 선언된 변수 used를 반환하는 getter 함수이다. 즉, 값이 저장된 요소의 개수를 반환한다. 함수 내에서 변수의 값을 바꾸면 안 되므로 함수 선언문 끝에 const를 붙여주었다.

(17). void reserve(size_t n)

```
void MyDoubleVector::reserve(size_t n)
{
    // postcondition: Extends the capacity of the allocated storage space for the elements of the vector container.
    double *larger_array;
    if (n == space)
        return;
    if (n < used)
        n = used;
    larger_array = new double[n];
    copy(data, data + used, larger_array);
    delete[] data;
    data = larger_array;
    space = n;
}
```

<그림 19. MyDoubleVector 클래스의 reserve 함수>

인자로 받은 n만큼 capacity를 늘리는 함수이다. capacity를 늘리기 위해 더 큰 메모리 공간을 할당해서 기존의 것을 지우고 새 공간을 가리키는 방법을 이용한다. 인자로 받은 n이 현재 capacity(space)와 같다면 함수를 종료하고 n이 채워져 있는 요소의 수, used 보다 작다면 n 값을 used로 바꾼다. n 크기만큼의 배열을 동적 할당하여 기존 데이터를 Deep copy로 복사한 후 기존 메모리 공간은 해제한다. 객체의 포인터 변수 data가 새 동적 배열을 가리키게 하고 변수 space를 n으로 바꾼다.

2. 각 함수 별 소스코드 설명

(18). bool empty() const

```
bool MyDoubleVector::empty() const
{
    // postcondition: if the vector container is empty, returns true.
    //                Otherwise, returns false.
    return used == 0;
}
```

<그림 20. MyDoubleVector 클래스의 empty 함수>

객체의 요소가 비어있는지 아닌지 확인하는 함수이다. used 값이 0이면 true를, 아니라면 false를 반환한다. 함수 내부에서 변수의 값을 바꾸면 안 되므로 선언문 끝에 const를 붙였다.

(19). void clear()

```
void MyDoubleVector::clear()
{
    // postcondition: All the elements of the vector are dropped
        delete[] data;
        used = 0;
        space = 0;
        data = new double[space];
}
```

<그림 21. MyDoubleVector 클래스의 clear 함수>

객체 요소의 모든 값들을 지우고 size와 capacity를 모두 0으로 초기화하는 함수이다. 포인터 변수 data가 가리키는 동적 배열을 해제하고 변수 used와 space를 0으로 만든다. 그리고 크기가 0인 동적 배열을 새로 할당해 data가 가리키게 한다.

3. 테스트 프로그램 코드 설명

소스코드	`MyDoubleVector v1;`
	`cout << "< initialize v1>" << endl << endl; // constructor test`
	`cout << "initial size of v1 : " << v1.size() << endl;`
	`cout << "initial capacity of v1 : " << v1.capacity() << endl << endl;`

실행 창	`< initialize v1>`
	
	`initial size of v1 : 0`
	`initial capacity of v1 : 1`

설명	Default constructor를 테스트하였다. 초기 size와 capacity를 출력하여 확인해보았다. 동시에 size 함수와 capacity 함수도 테스트하였다.

소스코드	`cout << "< v1.reserve(3) >" << endl; //reserve(size_t n) test`
	`v1.reserve(3);`
	`cout << "reserved capacity of v1 : " << v1.capacity() << endl << endl;`

실행 창	`< v1.reserve(3) >` `reserved capacity of v1 : 3`
설명	reserve 함수를 테스트하였다. 인자로 3을 주어 capacity가 3으로 변하는지 확인하였다.

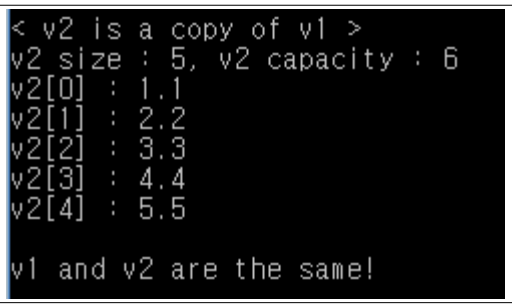
소스코드	`cout << "< v1.push_back [0] to [4] >" << endl; // push_back(double x) test`
	`double x = 1.1;`
	`for (i = 0; i < 5; i++)`
	`{`
	`  v1.push_back(x);`
	`  cout << "v1[" << i << "] : " << v1[i] << endl; // operator[] test`
	`  cout << "size : " << v1.size() << ", capacity : " << v1.capacity() << endl; // size(), capacity() test`
	`  x += 1.1;`
	`}`
	`cout << endl;`

실행 창	`< v1.push_back [0] to [4] >`
	`v1[0] : 1.1`
	`size : 1, capacity : 3`
	`v1[1] : 2.2`
	`size : 2, capacity : 3`
	`v1[2] : 3.3`
	`size : 3, capacity : 3`
	`v1[3] : 4.4`
	`size : 4, capacity : 6`
	`v1[4] : 5.5`
	`size : 5, capacity : 6`

설명	push_back 함수를 테스트하였다. double 값 x를 변화시켜가며 v1[0]부터 v1[4]까지 값을 넣어주었다. 이를 출력하여 확인하였고 이 과정에서 []연산자도 테스트하였다. push_back 함수를 사용할 때마다 변화하는 size와 capacity도 값을 출력하여 확인하였다.

3. 테스트 프로그램 코드 설명

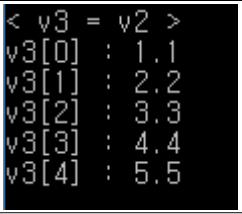
소스코드	`cout << "< v2 is a copy of v1 >" << endl; // copy constructor test`
	`MyDoubleVector v2(v1);`
	`cout << "v2 size : " << v2.size() << ", v2 capacity : " << v2.capacity() << endl;`
	`for (i = 0; i < v2.size(); i++)`
	`    cout << "v2[" << i << "] : " << v2[i] << endl;`
	`cout << endl;`
	
	`if (v1 == v2) // operator== test`
	`    cout << "v1 and v2 are the same!" << endl << endl;`

실행 창	
설명	Copy constructor를 테스트하였다. 객체 v2를 생성할 때 v1을 인자로 넣어 생성하였다. 그리고 v2의 size, capacity, 각 요소의 값들을 출력해 확인하였다. 객체 v1과 v2가 같은지는 == 연산자를 이용하여 확인하였다. 결과에 문구가 출력되었으므로 v1과 v2가 같고 copy constructor와 ==연산자가 잘 만들어졌음을 확인하였다.

소스코드	`double scalar = v1 * v2; // operator* test`
	`cout << "scalar product of v1 and v2 : " << scalar << endl << endl;`

실행 창	
설명	*연산자를 테스트하였다. 객체 v1와 v2를 *연산하여 그 값을 scalar 변수에 저장하였다. scalar 변수를 출력하여 내적이 잘 계산되었음을 확인하였다.

소스코드	`cout << "< v3 = v2 >" << endl; // operator= test`
	`MyDoubleVector v3 = v2;`
	`for (i = 0; i< v3.size(); i++)`
	`    cout << "v3[" << i << "] : " << v3[i] << endl;`
	`cout << endl;`

실행 창	
설명	=연산자를 테스트하였다. 객체 v3를 생성할 때 =연산으로 객체 v2를 넣어주었다. 바로 v3의 각 요소들을 출력하여 객체 v2의 요소들의 값과 같은지 확인하였다.

3. 테스트 프로그램 코드 설명

소스코드	`cout << "< v3 = v1 + (-v2) >" << endl; // operator+, unary operator-, operator= test`
	`v3 = v1 + (-v2);`
	`for(i = 0; i< v3.size(); i++)`
	`  cout << "v3[" << i << "] : " << v3[i] << endl;`
	`cout << endl;`

실행 창	`< v3 = v1 + (-v2) >`
	`v3[0] : 0`
	`v3[1] : 0`
	`v3[2] : 0`
	`v3[3] : 0`
	`v3[4] : 0`

설명	이항연산자 +와 단항연산자 -, 대입연산자를 테스트하였다. 객체 v2에 - 단항연산을 하여 요소들의 값을 음수로 바꾸고 객체 v1과 +연산을 하여 그 결과를 대입연산자로 객체 v3에 들어가게 하였다. v1과 v2의 요소들의 값이 같으므로 v3의 요소들이 0으로 바뀌었다.

소스코드	`cout << "< v3(5.5) : makes every element be 5.5 >" << endl; // operator() test`
	`v3(5.5);`
	`for (i = 0; i< v3.size(); i++)`
	`  cout << "v3[" << i << "] : " << v3[i] << endl;`
	`cout << endl;`

실행 창	`< v3(5.5) : makes every element be 5.5 >`
	`v3[0] : 5.5`
	`v3[1] : 5.5`
	`v3[2] : 5.5`
	`v3[3] : 5.5`
	`v3[4] : 5.5`

설명	()연산자를 테스트하였다. 인자로 5.5를 주어 v3 요소들의 값을 모두 5.5로 만들었다. 각 요소들을 출력해서 0에서 5.5로 값이 변한 것을 확인하였다.

소스코드	`cout << "< v2 = v3 - v1 >" << endl; // binary operator- test`
	`v2 = v3 - v1;`
	`for (i = 0; i< v2.size(); i++)`
	`  cout << "v2[" << i << "] : " << v2[i] << endl;`
	`cout << endl;`

실행 창	`< v2 = v3 - v1 >`
	`v2[0] : 4.4`
	`v2[1] : 3.3`
	`v2[2] : 2.2`
	`v2[3] : 1.1`
	`v2[4] : 0`

설명	이항연산자 -를 테스트하였다. 요소의 값들이 모두 5.5인 객체 v3와 v1을 -연산하여 반환 값을 v2에 대입하였다. v2의 각 요소들을 출력하여 요소들의 뺄셈이 잘 되었음을 확인하였다.

3. 테스트 프로그램 코드 설명

소스코드	`cout << "< v1 += v2 >" << endl; // operator+= test`
	`v1 += v2;`
	`cout << "size of v1 : " << v1.size() << ", capacity of v1 : " << v1.capacity() << endl;`
	`for (i = 0; i< v1.size(); i++)`
	`  cout << "v1[" << i << "] : " << v1[i] << endl;`
	`cout << endl;`

실행 창	`< v1 += v2 >`
	`size of v1 : 10, capacity of v1 : 10`
	`v1[0] : 1.1`
	`v1[1] : 2.2`
	`v1[2] : 3.3`
	`v1[3] : 4.4`
	`v1[4] : 5.5`
	`v1[5] : 4.4`
	`v1[6] : 3.3`
	`v1[7] : 2.2`
	`v1[8] : 1.1`
	`v1[9] : 0`

설명	+=연산자를 테스트하였다. v1와 v2를 +=연산하여 v1의 요소 끝에 v2가 덧붙여지도록 하였다. v1의 size와 capacity, 각 요소들을 출력하여 변화를 확인하였다.

소스코드	`cout << "< swap v1[8] and v1[9] >" << endl; // operator[] test`
	`double temp = v1[8];`
	`v1[8] = v1[9];`
	`v1[9] = temp;`
	`for (i = 0; i< v1.size(); i++)`
	`  cout << "v1[" << i << "] : " << v1[i] << endl;`
	`cout << endl;`

실행 창	`< swap v1[8] and v1[9] >`
	`v1[0] : 1.1`
	`v1[1] : 2.2`
	`v1[2] : 3.3`
	`v1[3] : 4.4`
	`v1[4] : 5.5`
	`v1[5] : 4.4`
	`v1[6] : 3.3`
	`v1[7] : 2.2`
	`v1[8] : 0`
	`v1[9] : 1.1`

설명	[]연산자를 테스트하기 v1[8]과 v1[9]의 값을 서로 바꾸었다.

3. 테스트 프로그램 코드 설명

소스코드	`cout << "< v1 pop_back 5 times >" << endl; // pop_back() test`
	`cout << "size before pop_back : " << v1.size() << endl;`
	`for (i = 0; i < 5; i++)`
	`  v1.pop_back();`
	`cout << "size after pop_back : " << v1.size() << endl;`
	`for (i = 0; i< v1.size(); i++)`
	`  cout << "v1[" << i << "] : " << v1[i] << endl;`
	`cout << endl;`
	
	`cout << "< v1 pop_back 5 times again>" << endl; // empty() test`
	`for (i = 0; i < 5; i++)`
	`  v1.pop_back();`
	`if (v1.empty())`
	`  cout << "v1 is empty" << endl << endl;`

실행 창	`< v1 pop_back 5 times >`
	`size before pop_back : 10`
	`size after pop_back : 5`
	`v1[0] : 1.1`
	`v1[1] : 2.2`
	`v1[2] : 3.3`
	`v1[3] : 4.4`
	`v1[4] : 5.5`
	
	`< v1 pop_back 5 times again>`
	`v1 is empty`

설명	pop_back 함수와 empty 함수를 테스트하였다. 먼저 v1 객체의 pop_back를 5번 호출하여 v1[5]부터 v1[9]까지의 요소를 삭제하였다. size 값과 요소를 출력하여 size가 줄어들었음을 확인하였다. 그리고 pop_back을 5번 더 해주어 배열을 완전히 비운 뒤 empty 함수로 배열이 비어있는지 확인 작업을 하였다.

3. 테스트 프로그램 코드 설명

소스코드	cout << "< clearing v1 >" << endl; // clear() test
	v1.push_back(10.1);
	v1.push_back(10.2);
	v1.push_back(10.3);
	cout << "v1 contains";
	for (i = 0; i < v1.size(); i++)
	cout << " " << v1[i];
	cout << endl;
	cout << "-- clear & push_back --" << endl;
	v1.clear();
	v1.push_back(20.1);
	v1.push_back(20.2);
	cout << "v1 contains";
	for (i = 0; i < v1.size(); i++)
	cout << " " << v1[i];
	cout << endl;

실행 창	< clearing v1 > v1 contains 10.1 10.2 10.3 -- clear & push_back -- v1 contains 20.1 20.2
설명	clear 함수를 테스트하였다. push_back 함수로 v1의 요소를 채운 뒤 clear 함수를 호출하고 push_back으로 요소를 다시 채웠다. 이를 출력한 결과를 토대로 clear 함수 호출로 처음에 채웠던 값들이 사라졌음으로 확인할 수 있었다.

4. 고 찰

수업 시간에 동적 배열을 이용하여 Bag 클래스를 구성하는 것을 배웠기 때문에 조금 수월하게 과제를 수행할 수 있지 않았나 싶다. 강의 자료에 나와 있는 몇 가지 유사한 함수들을 응용하여 설계했더니 큰 오류 없이 잘 실행이 되었다. 과제에 따로 제시되지 않은 부분들은 최대한 STL의 vector 클래스와 유사하게 만들려고 노력하였다. STL vector 관련 문서들을 많이 참고하고 공부하여 활용하였다. 물론, 메모리의 크기를 늘리고 줄이는데 있어 STL의 vector 클래스처럼 만드는 게 어느 정도 한계가 있었다. 그래도 개인적으로 기대했던 수준의 vector 클래스가 잘 만들어진 것 같고, 이전에 배웠던 것들을 복습할 수 있는 좋은 기회였다.

조금 아쉽고 어려웠던 부분으론 코드를 작성하여 실행해보는 중간 중간 원인을 알 수 없는 Runtime 에러가 종종 발생했다는 것이다. 아직 메모리를 할당하고 해제하는데 있어 익숙하지 않은 탓인지 발생한 에러의 원인을 정확히 집어내는데 어려움을 겪었다. Google 검색을 통해 원인을 알아내서 해결한 경우도 있었지만 원인도 모른 채 그냥 여러 방법으로 코드를 짜다 해결한 경우도 있었다. 특히, 메모리를 해제하는 부분과 std::copy 함수 사용이 마음처럼 잘 되지 않았는데 delete[]와 destructor, copy 부분에 대해 공부가 더 필요한 것 같다.