제출일 : 2022-10-07

KWANGWOON UNIVERSITY

---

---


| 담당교수 | 김용혁 교수님 | 학 번 | 2021203034 |
|---|---|---|---|
| 학 과 | 소프트웨어학부 | 이 름 | 허찬영 |

KWANGWOON UNIVERSITY

**my environment : Visual Studio 2022 (Windows)**

■ Begin with a summary of my results

```
class MyDoubleVector {
        public:
                ...
        private:
                double *data;
                ...
        };
```

Make class *MyDoubleVector* similar to class *vector<double>* in STL and also write a test program to check that all the member functions/operators of your class *MyDoubleVector* work correctly.

■ For each mission, explain how I completed it

○ MydoubleVector.h and MydoubleVector.cpp

```
private:
    size_t sz; // 크기
    double* data; // 요소를 가리키는 포인터
    size_t init_capacity; // 할당된 영역의 크기
```

MyDoubleVector class의 private 영역에 세 가지 멤버 변수를 정의했다.

size_t 타입의 sz 변수는 데이터가 저장되어 있는 요소의 개수를 저장한다.

double* 타입의 data 변수는 요소를 가리키는 포인터를 저장한다.

init_capacity 변수는 메모리에 할당된 영역의 크기를 저장한다.

*** CONSTRUCTORS ***

```
MyDoubleVector::MyDoubleVector()
    :sz{ 0 }, data{ new double[Default_Capacity] }, init_capacity{ Default_Capacity }
{}

MyDoubleVector::MyDoubleVector(const MyDoubleVector& v)
    :sz{ v.sz }, data{ new double[v.init_capacity] }, init_capacity{ v.init_capacity }
{
    for(size_t i =0; i<v.sz; ++i)
        data[i] = v.data[i];
}

MyDoubleVector::~MyDoubleVector() // Desturctor
{
    delete[] data;
}
```

```
static const size_t Default_Capacity = 100; // vector의 기본 크기
```

default constructor로 객체를 초기화 했다. data는 new 연산자를 통해 메모리에 Defualt_Capacity 값만큼 할당했다. 깊은 복사를 위해 copy constructor를 정의 하였고 복사하고자 하는 객체의 요소들의 포인터를 복사했다. 그리고 destructor를 통해 객체가 사라지면 할당한 메모리를 반환하도록 구현했다.

(수정)

```
MyDoubleVector::MyDoubleVector(const MyDoubleVector& v)
    :sz{ v.sz }, data{ new double[v.init_capacity] }, init_capacity{ v.init_capacity }
{
    copy(v.data, v.data + v.sz, data);
    //for(size_t i =0; i<v.sz; ++i)
    //  data[i] = v.data[i];
}
```

복사를 진행 하는 방식을 copy 함수를 이용한 방식으로 바꿨다. (복사 생성자, operator=, operator+=에서 변경)

*** OPERATORS ***

```
void MyDoubleVector::operator+=(const MyDoubleVector& v) {
    double* newPtr = new double[v.sz + sz];
    for (size_t i = 0; i < sz; i++) {
        newPtr[i] = data[i];
    }
    for (size_t i = sz; i < sz + v.sz; ++i) {
        newPtr[i] = v.data[i - sz];
    }
    delete[] data;
    this->data = newPtr;
    this->sz = v.sz + sz;
    this->init_capacity = v.init_capacity *2;
}
```

operator+=는 오른쪽 객체를 왼쪽 객체 뒤에 덧붙이는 기능을 한다.

그렇기 때문에 원래 객체의 크기에 덧붙이고자 하는 객체의 크기만큼 메모리에 할당을 하고 data

를 새로 할당한 객체에 복사를 했다. 그리고 원래 객체를 해제하고 새로 만든 객체를 이용하여 변수를 재정의했다.

```
MyDoubleVector& MyDoubleVector::operator=(const MyDoubleVector& v) {
    if (this == &v)
        return *this;

    double* newPtr = new double[v.sz];
    for (size_t i = 0; i < v.sz; i++) {
        newPtr[i] = v[i];
    }
    delete[] data;
    data = newPtr;
    sz = v.sz;
    init_capacity = v.init_capacity;

    return *this;
}
```

operator= 객체는 복사 대입 연산자로 객체를 복사하는 기능을 한다.

만약 두 객체가 동일할 경우 아무런 작업을 하지 않는다. 두 객체가 동일하지 않을 경우 새로운 객체를 할당하고 모든 요소를 하나 하나씩 복사한 뒤 오래된 객체를 해제 하는 방식으로 구현했다. 그리고 = 연산자를 한 문장에 여러 번 쓸 수 있기 때문에 자기 참조 반환을 이용했다.

```
double MyDoubleVector::operator*(const MyDoubleVector& v) {
    assert(this->size() == v.size());

    double sum = 0;
    MyDoubleVector product(*this);
    for (int i = 0; i < product.size(); i++)
        product.data[i] *= v.data[i];
    for (int i = 0; i < product.size(); i++) {
        sum += product.data[i];
    }
    return sum;
}
```

다음은 operator*이다. 내적 값이 나오도록 구현했다.

```
MyDoubleVector MyDoubleVector::operator+(const MyDoubleVector& v) {
    assert(this->size() == v.size());

    MyDoubleVector sum(*this);
    for (int i = 0; i < sum.size(); i++)
        sum.data[i] += v.data[i];
    return sum;
}

MyDoubleVector MyDoubleVector::operator-(const MyDoubleVector& v) {
    assert(this->size() == v.size());

    MyDoubleVector diff(*this);
    for (int i = 0; i < diff.size(); i++)
        diff.data[i] -= v.data[i];
    return diff;
}

MyDoubleVector MyDoubleVector::operator*(const MyDoubleVector& v) {
    assert(this->size() == v.size());

    MyDoubleVector product(*this);
    for (int i = 0; i < product.size(); i++)
        product.data[i] *= v.data[i];
    return product;
}
```

다음은 operator+, operator-, operator* 연산자이다. assert 함수를 이용하여 잘못된 객체가 들어올 경우 오류 메시지를 출력하고 프로그램의 작동을 끝낸다. 올바른 객체일 경우 새로운 객체를 생성하고 요소에 덧셈 연산을 한 뒤, 새로운 객체를 반환한다. operator-, operator*도 이와 같은 방식이다.

```
MyDoubleVector MyDoubleVector::operator-() {
    MyDoubleVector minus(*this);
    for (int i = 0; i < sz; i++) {
        if (minus.data[i] == 0);
        else
            minus.data[i] *= -1;
    }
    return minus;
}
```

부호를 바꾸는 operator- 연산자이다. 앞서 설명했던 operator-(const MyDoubleVector&)과 매개변수가 다르기 때문에 다른 기능을 하는 연산자이다. 새로운 객체를 생성하고 요소의 값이 0이 아닐 경우 요소 하나 하나에 -1를 곱하는 연산을 한 뒤 새로운 객체를 반환한다.

```
bool operator==(const MyDoubleVector& v1, const MyDoubleVector& v2) {
    if (v1.size() == v2.size()) {
        for (int i = 0; i < v1.size(); i++) {
            if (v1[i] != v2[i]) {
                return false;
            }
        }
        return true;
    }
    else
        return false;
}
```

두 객체가 동일한지 비교하는 operator== 연산자이다. 두 객체의 크기가 같으면 작동하고 아니면 false를 반환한다. 그리고 요소 하나 하나를 비교하여 모든 요소가 같을 경우 true를 반환하고 하나라도 다르면 false를 반환한다.

```
MyDoubleVector& MyDoubleVector::operator()(int n) {
    for (int i = 0; i < sz; i++) {
        this->data[i] = n;
    }
    return *this;
}
```

모든 요소를 사용자가 지정한 값으로 초기화 하는 operator()이다. int n을 매개변수로 받아 모든 요소에 n을 대입한 뒤 *this를 반환한다.

```
double& MyDoubleVector::operator[](int idx) {
    assert(idx >= 0 && idx < sz);

    return data[idx];
}

double& MyDoubleVector::operator[](int idx) const {
    assert(idx >= 0 && idx < sz);

    return data[idx];
}
```

특정한 인덱스의 요소를 반환하는 operator[]이다. 잘못된 index에 접근할 경우 assert 함수로 처리를 했다. const 객체의 index 접근을 허용하기 위해 const 버전의 operator[]도 구현했다.

*** MEMBER FUNCTION ***

```
size_t MyDoubleVector::capacity() const {
    return init_capacity;
}

size_t MyDoubleVector::size() const {
    return sz;
}
```

각각 init_capacity와 sz의 값을 반환한다.

```
void MyDoubleVector::pop_back() {
    if (!this->empty()) {
        data[sz - 1] = '\0';
        sz--;
    }
}

void MyDoubleVector::push_back(double x) {
    if (init_capacity == 0)
        reserve(8); // 8개의 요소를 저장할 수 있는 공간으로 시작
    else if (sz == init_capacity)
        reserve(2 * init_capacity);
    data[sz] = x;
    sz++;
}
```

pop_back 함수는 요소의 크기가 0이 아니라면 가장 마지막 요소를 널 문자로 초기화하고 sz의 값을 1만큼 감소시키는 함수다. push_back 함수는 init_capacity가 0이라면 reserve(8)을 호출하여 메모리를 sizeof(double)*8만큼 할당하고 sz와 init_capacity가 같으면 메모리를 sizeof(double)*2*init_capacity만큼 할당한다. 그리고 sz 위치에 x를 넣고 sz 크기를 1만큼 늘려 새로운 요소를 저장한다.

```
void MyDoubleVector::reserve(size_t n) {
    if (n <= init_capacity) return;
    double* newPtr = new double[n];
    for (size_t i = 0; i < sz; i++)
        newPtr[i] = data[i];
    delete[] data;
    data = newPtr;
    init_capacity = n;
}
```

reserve 함수는 메모리를 추가적으로 할당하는 기능을 한다. 매개변수로 받은 크기만큼 새로운 객

체를 생성한 뒤 새로운 객체에 원래 요소를 복사한다. 그리고 오래된 객체를 해제하고 data를 새로운 객체를 가리키도록 구현했다.

```
bool MyDoubleVector::empty() const {
    if (sz == 0)
        return true;
    else
        return false;
}

void MyDoubleVector::clear() {
    delete[] data;

    double* newData = new double[0];
    data = newData;
    sz = 0;
}
```

empty() 함수는 sz가 0이면 true를 반환, 아니면 false를 반환한다.

clear() 함수는 원래 객체를 해제 하고 0크기의 새로운 객체를 생성하여 data가 새로운 객체를 가리키도록 구현했다.

○ HW1_2021203034.cpp

```
// push_back test
for (size_t i = 0; i < 10; i++) {
    v1.push_back(i);
}
for (size_t i = 10; i < 21; i++) {
    v2.push_back(i);
}
```

push_back을 테스트하는 코드.

```
// operator test and index operator test
opt = v1 + v2;
cout << "operator+ test\n";
for (int i = 0; i < opt.size(); i++) {
    cout << opt[i] << '\n';
}
opt = v1 - v2;
cout << "operator- test\n";
for (int i = 0; i < opt.size(); i++) {
    cout << opt[i] << '\n';
}
opt = v1 * v2;
cout << "operator* test\n";
for (int i = 0; i < opt.size(); i++) {
    cout << opt[i] << '\n';
}
opt = -opt;
cout << "operator- test\n";
for (int i = 0; i < opt.size(); i++) {
    cout << opt[i] << '\n';
}
opt(7);
cout << "operator() test\n";
```

여러가지 operator를 테스트 하는 코드(모든 operator 코드를 테스트)

```
// size and capacity test
cout << "size and capacity test: " << v1.size() << ", " << v2.size() << '\n';

// pop_back test
v2.pop_back();
cout << "size and capacity test after pop_back: " << v1.size() << ", " << v2.size() << '\n';

// copy constructor test
copy = v1;

// clear and empty test
copy.clear();
cout << "empty test: " << copy.empty() << endl;
for (int i = 0; copy.size(); i++) {
    cout << copy[i] << '\n';
}
```

여러가지 함수들을 테스트 하는 코드

모두 정상적으로 작동한다.

■ Conclude with some comments on my work

이 과제를 하면서 STL에 있는 vector에 대해서 조금 더 이해할 수 있었다.

다양한 operator overloading를 구현하면서 연산자 오버로딩을 이해할 수 있었다.

메모리를 할당하고 해제하는 연습을 할 수 있었다.

class를 구현해볼 수 있었다.