Data Structures 2020

MyIntVector.cpp/h

HW1_2016726028

이민재 | 자료구조 | 10/13/20

MyIntVector.h - Private Members

```
private:
    int *data;
    static const size_t Default_Capacity = 100; //디폴트 크기 : 100
    size_t v_capacity; //MyIntVector 크기 최대값(용량)
    size_t v_used; //MyIntVector 현재 크기(사용량)
};
```

MyIntVector 의 private 멤버들은 컨테이너의 사용량, 총 크기 , 자료형을 정의하는 v_used, v_capacity, data 로 구성되어 있다. Default_Capacity 의 크기는 100 으로 사전 정의 없이 기본 생성자로 생성하였을 때 , 적용되는 v_capacity 의 값이다.

MyIntVector.cpp - Implemented Members

Default Constructor : MyIntVector(size_t init_capacity = Default_Capacity);

```
MyIntVector(size_t init_capacity = Default_Capacity); //default constructor
//precondition: init_capacity(integer) should be more than 0
//postcondition: init_capacity - > v_capacity, if init_capacity N/A.. Default_Capacity -> v_capacity
//set the size of array (data), allocate into HEAP, initialize v_used = 0; v_capacity= ..
```

디폴트 생성자는 인자를 받은 정수형에 따라 int 배열을 힙 메모리에 동적 할당한다. 이때, 초기 크기는 0 보다 큰 정수값 이여야 하며,(precnd) 인자로 받은 capacity 값이 있다면 이에 맞추어 v_capcity 값을 설정하여 객체를 생성한다. (없을경우 default 크기인 100) (postcnd)

```
MyIntVector::MyIntVector(size_t init_capacity)
{
    data = new int[init_capacity];
    v_capacity = init_capacity;
    v_used = 0;
}
```

입력받은 capacity 인자의 값에 따라 data 배열의 크기를 설정 한 뒤, 힙메모리에 할당하고 , v_used 값을 0 으로 초기화시킨다.

Result:

```
void printVector(string name ,MyIntVector& v) {
        cout << endl;
        cout << "MyIntVector " << name << endl;
        cout << "capacity : " << v.capacity() << endl;
        cout << "size : " << v.size() << endl << endl;
}
void printElem(string name, MyIntVector& v) {
        cout << name <<"[0] : " << v[0] << endl;
        cout << name <<"[1] : " << v[1] << endl;
        cout << name << "[2] : " << v[2] << endl << endl;
}
```

테스트를 위해 벡터 내부 정보출력을 하는 함수 2 개를 생성하였다.

printVector 는 해당 벡터의 이름 , 크기 , 사용량을 출력하고

printElem 은 해당 벡터의 원소들을 인덱스 값을 참고하여 출력한다.

```
cout << ">>Test Default constructor" << endl << endl;
cout << "MyIntVector v1;" << endl;
MyIntVector v1;

printVector("v1", v1);
cout << "MyIntVector v2(50);" << endl;
MyIntVector v2(50);
printVector("v2", v2);
```

```
>>Test Default constructor

MyIntVector v1;

MyIntVector v1
capacity : 100
size : 0

MyIntVector v2(50);

MyIntVector v2
capacity : 50
size : 0
```

V1 은 capacity 인자 값을 설정하지 않고 선언 한 반면, v2 는 capacity 를 50 으로 설정하고 선언 하였다. 그 결과 오른쪽 화면에서 처럼 v1 의 capacity 는 100 ,v2 의 capacity 는 50 으로 설정된 것을 확인 가능하였다.

Destructor : ~MyIntVector();

```
~MyIntVector(); //Destructor
//preconditon: None
//postcondition: delete vector data & return HEAP data resource
```

소멸자로써 내부 data 를 delete 하고 메모리를 반환하는 기능을 수행한다.

```
MyIntVector::~MyIntVector()
{
    delete[] data;
}
```

이를 위해 delete[]를 사용하였다.

Result:

프로그램이 오류로 인해 종료되거나 , 메인 함수가 종료되는 경우. 디버깅 화면에서 소멸자가 호출되는 것을 확인 가능하였다.

Copy Constructor: MyIntVector(const MyIntVector& v);

```
MyIntVector(const MyIntVector& v); //Copy constructor for deep copy
//precondition: target Vector should be different with original Vector (copy v1 to v2,  not in v1)
//postcondition: make same structure with v(copy of v)
//v1 data(original) - > v2 data (target) , initialize with same value
```

복사 생성자는 자기 자신이 아닌 다른 MyIntVector 객체를 받아야 하며 (precnd) 그 인자로 받은 v 와 같은 내용으로 복사하여 객체를 새로 생성한다. (postcnd)

```
MyIntVector::MyIntVector(const MyIntVector& v)
{
    data = new int[v.v_capacity];
    v_capacity = v.v_capacity;
    v_used = v.v_used;
    std::copy(v.data, v.data + v_used, data);
}
```

인자로 전달받은 v 의 멤버 변수 data 는 힙 메모리 접근하는 주소값을 가지고 있기 때문에 이를 복사하기 보단, algorithm 헤더 기능 중 하나인 copy 를 사용하여 data 값을 복사한다. 나머지 멤버 변수인 v_capcity 와 v_used 는 v 값과 같게 복사한다. const 형으로 받기 때문에 전달받은 v 의 멤버 변수값이 바뀌는 일은 없다.

Result:

```
cout << ">>push back(1),(2),(3) into v2" << endl << endl;
v2.push_back(1);
v2.push_back(2);
v2.push_back(3);
printVector("v2", v2);

printElem("v2", v2);
cout << "----------------------" << endl;
cout << ">>Test Copy constructor" << endl << endl;
cout << "MyIntVector v6 = v2" << endl;
MyIntVector v6 = v2;
printVector("v6", v6);
printElem("v6", v6);
cout << "----------------------" << endl;
```

```
----------------------
>>push back(1),(2),(3) into v2


MyIntVector v2
capacity : 50
size : 3

v2[0] : 1
v2[1] : 2
v2[2] : 3

----------------------
>>Test Copy constructor

MyIntVector v6 = v2

MyIntVector v6
capacity : 50
size : 3

v6[0] : 1
v6[1] : 2
v6[2] : 3

----------------------
```

Default 생성자 예시에서 이어지는 main 함수 코드로, v2(50) 으로 생성된 v2 에 push_back 기능으로 1,2,3 을 삽입한 뒤, v6 를 복사 생성자를 이용하여 생성하였다. V6 가 v2 와 내용이 같음을 확인 가능하다.

Assignment operator : MyIntVector& operator=(const MyIntVector& v);

```
MyIntVector& operator=(const MyIntVector& v); //Assignment operator (=) for deep copy
//precnd: obj should be MyIntVector Type
//postcnd: Original vector's data turn into target vector's data.
// v1 = v1 should return v1
```

MyIntVector 객체를 인자로 받는다.(precnd) 그 객체와 똑 같은 정보로 original vector 을 복사하며 복사 생성자와는 달리 새로운 객체 생성을 하지는 않는다. 자기 자신이 인자로 들어올 경우 자기 자신을 리턴한다.(postcnd)

```
MyIntVector& MyIntVector::operator=(const MyIntVector& v)
{
    if (this == &v) return *this;
    if (v_capacity != v.v_capacity)
    {
        delete[] data;
        data = new int[v.v_capacity];
        v_capacity = v.v_capacity;
    }
    v_used = v.v_used;
    std::copy(v.data, v.data + v_used, data);
    return *this;
}
```

기존에 할당한 data 를 delete 하고 인자로 받은 v 와 동일한 size, capcity 를 설정한 후 힙 메모리에 할당한다. 역시 copy 기능을 사용하여 data 의 각 인덱스에 맞게 초기화 시킨다.

Result:

```
cout << ">>Test = Operator" << endl << endl;
cout << "v1 = v2" << endl;
v1 = v2;

printVector("v1", v1);
printElem("v1", v1);
cout << "----------------------" << endl;
```

```
----------------------
>>Test = Operator

v1 = v2

MyIntVector v1
capacity : 50
size : 3

v1[0] : 1
v1[1] : 2
v1[2] : 3

----------------------
```

위 예시에서 이어지는 main 함수 코드로, v2(50) 으로 생성된 v2 에 push_back 기능으로 1,2,3 을 삽입한 뒤, v1 에 v2 를 복사 하였다. (v1 = v2) 그 후 v1 의 정보와 원소들을 출력한 화면이다. V2 와 capacity, size, data 들이 동일하게 복사되었음을 출력값 으로 확인할 수 있다.

Operator += : void operator+=(const MyIntVector& v);

```
void operator+=(const MyIntVector& v); //Operator: += : Appends RHS object to LHS one
//precnd: obj should be MyIntVector Type
//postcnd: put target vector(v2) into original vector(v1)'s back
//capacity of new v1 is 'v_used of v1 + v_used of v2' if v_capcity is smaller than that
```

MyIntVector 가 객체로 들어와야한다.(precnd) 인자로 받은 v 를 현재 대상(original)에 뒤에 덧붙인다.(postcnd)

```
void MyIntVector::operator+=(const MyIntVector& v)
{
    if (v_capacity < v_used + v.v_used)
        reserve(v_used + v.v_used);
    std::copy(v.data, v.data + v.v_used, data + v_used);
    v_used += v.v_used;
}
```

만약 original 의 사용량과 덧붙인 벡터의 사용량의 합이 original 의 capacity 보다 크다면 벡터 총크기보다 사용량이 많아짐으로 reserve 를 사용하여 각 사용량의 합으로 벡터의 총 capacity 를 늘려주었다.

Result:

```
cout << ">>Test += Operator" << endl << endl;
cout << "v1 += v2" << endl;
v1 += v2;
printVector("v1", v1);
printElem("v1", v1);
printElem2("v1", v1);
cout << "----------------------" << endl;
```

```
>>Test += Operator

v1 += v2

MyIntVector v1
capacity : 50
size : 6

v1[0] : 1
v1[1] : 2
v1[2] : 3

v1[3] : 1
v1[4] : 2
v1[5] : 3

----------------------
```

역시 이전 예제에서 이어지는 코드로 , v1 벡터의 뒤에 v2 를 += 연산자를 사용하여 넣었다.
오른쪽화면은 v1 의 뒤에 3,4,5 인덱스로 v2 의 data 가 삽입된 것을 출력 값으로 확인할 수 있다.

Operator [ ] : int& operator[](size_t index);

```
int& operator[](size_t index); // Operator[] :
//precnd: index should be in '0<= index < v_used'
//postcnd: return reference of data with index
//If the requested position is out of range, Print some messages(error) and terminate the program.
```

```cpp
int& MyIntVector::operator[](size_t index)
{
    if (!(index >= 0 && index < v_used))
        throw std::out_of_range("input index is out of range");

    return data[index];
}
```

Result:

```cpp
printVector("v1", v1);
printElem("v1", v1);
printElem2("v1", v1);
cout << "v1[6] : " << v1[6] << endl;
```

```
----------------------
>>Test += Operator

v1 += v2

MyIntVector v1
capacity : 50
size : 6

v1[0] : 1
v1[1] : 2
v1[2] : 3

v1[3] : 1
v1[4] : 2
v1[5] : 3

Error : input index is out of range
```

printElem 을 모든 예시에서 활용하기 때문에 operator []를 따로 테스트 하기보단, 오류상황에서 오류 메시지를 출력하는 지를 검증하였다. 위의 예시에서 존재하지 않는 v1[6]를 참조하려 할 경우 에러메시지로 out of range error 문이 출력됨을 확인 할 수있다.

Operator + : MyIntVector operator+(const MyIntVector& v);

```cpp
MyIntVector operator+(const MyIntVector& v); //Binary operator +
//precnd:obj should be MyIntVector Type, applicable only when the sizes of the two operands is the same
//postcnd: add data in each vector index by index
//Returns an object that is a vector-sum of the two operand objects.
```

역시 MyIntVector 가 객체로 들어와야한다. 또한 양 항의 벡터의 size(v_used) 가 동일하여야 연산이 성립한다.(precnd)

```cpp
MyIntVector MyIntVector::operator+(const MyIntVector& v)
{
    if (v_used != v.v_used)
        throw std::length_error("vector sizes is not same");

    MyIntVector temp(*this);
    temp.reserve(v_used);
    for (int i = 0; i < v_used; i++)
        temp.data[i] += v.data[i];
    return temp;
}
```

operator+은 현재 vector 와 v 의 vector 를 덧셈 연산을 한 뒤 그 값을 새로운 객체에 저장하여 그 값을 반환한다.

Result:

```
cout << "----------------------" << endl;
cout << ">>Test + Operator" << endl << endl;
MyIntVector v3;
cout << "v3 = v1 + v2" << endl;
v3 = v1 + v2;
printElem("v1", v1);
printElem("v2", v2);
printVector("v3", v3);
printElem("v3", v3);
```

역시 이전 예제들과 이어지는 코드로 v1 + v2 로 각 data 들을 인덱스 별로 각각 더하였으며 이렇게 새로 만들어진 컨테이너구조를 를 v3 에 할당하여 그 내용을 출력한 것이 오른쪽 결과이다.

```
----------------------
>>Test + Operator

v3 = v1 + v2
v1[0] : 1
v1[1] : 2
v1[2] : 3

v2[0] : 1
v2[1] : 2
v2[2] : 3


MyIntVector v3
capacity : 3
size : 3

v3[0] : 2
v3[1] : 4
v3[2] : 6

----------------------
```

Operator – (Binary) : MyIntVector operator-(const MyIntVector& v);

```
MyIntVector operator-(const MyIntVector& v); //Binary operator -
//precnd: obj should be MyIntVector Type, applicable only when the sizes of the two operands is the same
//postcnd: subtract data in each vector index by index
//Returns an object that is a vector-difference of the two operand objects.
```

역시 MyIntVector 가 객체로 들어와야한다. 또한 양 항의 벡터의 size(v_used) 가 동일하여야 연산이 성립한다.(precnd)

```
MyIntVector MyIntVector::operator-(const MyIntVector& v)
{
    if (v_used != v.v_used)
        throw std::length_error("vector sizes is not same");
    MyIntVector temp(*this);
    temp.reserve(v_used);
    for (int i = 0; i < v_used; i++)
        temp.data[i] -= v.data[i];

    return temp;
}
```

operator-은 뺄셈 연산을 한 뒤 그 값을 새로운 객체에 저장하여 그 값을 반환한다.

Result:

```
cout << ">>Test - Operator" << endl << endl;
MyIntVector v4(50);
cout << "v4 = v1 - v2" << endl;
v4 = v1-v2;
printElem("v1", v1);
printElem("v2", v2);
printVector("v4", v4);
printElem("v4", v4);
cout << "----------------------" << endl;
```

```
----------------------
>>Test - Operator

v4 = v1 - v2
v1[0] : 1
v1[1] : 2
v1[2] : 3

v2[0] : 1
v2[1] : 2
v2[2] : 3


MyIntVector v4
capacity : 3
size : 3

v4[0] : 0
v4[1] : 0
v4[2] : 0

----------------------
```

이전 예제들과 이어지는 코드로 v1 - v2 로 각 data 들을 인덱스 별로 각각 뺄셈하였으며 이렇게 새로 만들어진 컨테이너구조를 를 v4 에 할당하여 그 내용을 출력한 것이 오른쪽 결과이다.

Operator * : int operator*(const MyIntVector& v);

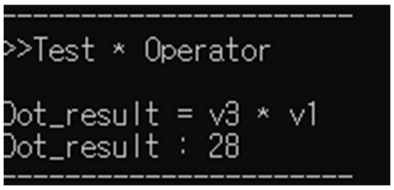
```
int operator*(const MyIntVector& v); //Binary operator *
//precnd: obj should be MyIntVector Type, applicable only when the sizes of the two operands is the same
//postcnd: multiply data in each vector index by index , make Dot product of vectors
//Returns the scalar product (or dot product) value of the two operand objects.
```

역시 MyIntVector 가 객체로 들어와야한다. 또한 양 항의 벡터의 size(v_used) 가 동일하여야 연산이 성립한다.(precnd) 위 두 연산과는 다르게 레퍼런스로 반환을 하지 않고 ,int 형 결과값을 가진다.

```
int MyIntVector::operator*(const MyIntVector& v)
{
    if (v_used != v.v_used)
        throw std::length_error("vector sizes is not same");

    int DOT = NULL;

    for (int i = 0; i < v_used; i++)
        DOT += this->data[i] * v.data[i];

    return DOT;
}
```

operator*은 내적 연산을 한 뒤 그 값을 반환한다.

Result:




```
cout << ">>Test * Operator" << endl << endl;
int Dot_result;
cout << "Dot_result = v3 * v1" << endl;
cout << "Dot_result : " << v3 * v1 << endl;
cout << "----------------------" << endl;
```

V3 와 v1 을 v3* v1 의형식으로 계산하여 결과값을 출력한 화면이다. 두 벡터의 내적값이 (2,4,6) * (1,2,3) = (2+8+18) = 28 정상적으로 출력됨을 확인 할 수 있다.

Operator – (Unary) : MyIntVector operator-();

```
MyIntVector operator-(); //Unary operator -
//precnd: None
//postcnd: store every elements in negative ver. (new obj)
//Returns an object of which each element is the unary negation of the corresponding element in the operand object
```

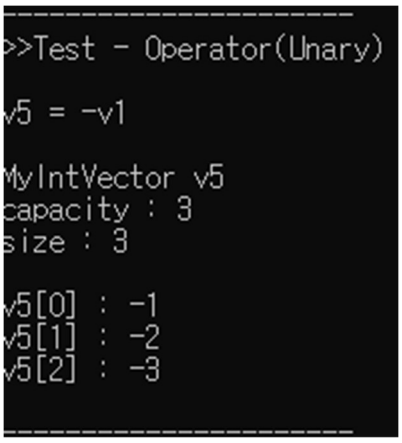
부호를 반전시킨 값을 새로운 객체에 저장해야한다(postcnd)

```
MyIntVector MyIntVector::operator-()
{
    MyIntVector temp(*this);
    temp.reserve(v_used);
    for (int i = 0; i < v_used; i++)
        temp.data[i] *= -1;

    return temp;
}
```

현재 객체의 data 값들의 부호를 반전시킨 객체를 인덱스별로 저장한 뒤, 반환한다.

Result:

```
cout << ">>Test - Operator(Unary)" << endl << endl;
MyIntVector v5;
cout << "v5 = -v1" << endl;
v5 = -v1;
printVector("v5", v5);
printElem("v5", v5);
cout << "----------------------" << endl;
```



-v1 의 값을 v5 에 저장시켜 출력한 결과물이다. 1, 2, 3 이 각각 부호 반전되어 -1, -2,-3 으로 저장된 것을 확인 할 수 있다.

Operator == : bool operator==(const MyIntVector& v);

```
bool operator==(const MyIntVector& v); // Binary operator ==
//precnd:  obj should be MyIntVector Type, size(v_used) of each vectors should be same
//postcnd: compare each vectors data, index by index
//Returns whether or not the two operand vectors are the same.
```

객체는 역시 MyIntVector 타입 이여야하고 각 항의 size 가 동일해야한다 (precnd) 인자로 넘겨받은 v 와 original 벡터의 각 data 를 인덱스별로 대조하고 값이 다른 것이 있는지 검출한다.

```
bool MyIntVector::operator==(const MyIntVector& v)
{
    if(v_used != v.v_used)
        throw std::length_error("vector sizes is not same");
    for (int i = 0; i < v_used; i++)
        if (this->data[i] != v.data[i])
            return false;
    return true;
}
```

현재 객체의 data 값들과 인자로 받은 v 객체의 data 값을 인덱스에 따라 각각 비교하여 하나라도 다를 경우 false 를 반환, 모두 다 같을 경우 true 를 반환하는 비교연산자이다.

Result:

```
cout << "----------------------" << endl;
cout << ">>Test == Operator" << endl << endl;
printElem("v1", v1);
printElem("v2", v2);
printElem("v3", v3);
cout << "v1 == v2 result: " << (v1==v2) << endl;
cout << "v3 == v1 result: " << (v3 == v1) << endl;
cout << "----------------------" << endl;
```

```
----------------------
>>Test == Operator

v1[0] : 1
v1[1] : 2
v1[2] : 3

v2[0] : 1
v2[1] : 2
v2[2] : 3

v3[0] : 2
v3[1] : 4
v3[2] : 6

v1 == v2 result: 1
v3 == v1 result: 0
----------------------
```

V1 == v2 와 v3 == v1 의 결과값을 출력해보았다. V1 은 v2 에서 복사되어 생성된 벡터임으로 data 가 완전히 같아 결과값으로 1(true)가 출력됨을 알 수 있고 v3 의 경우 v1+v2 의 연산으로 생성된 벡터임으로 data 값이 서로 달라 0(false)가 출력되는 것을 확인 할 수 있다.

Operator ( ) : void operator()(int element);

```
void operator()(int element); // Unary operator ()
//precnd: factor should be integer
//postcnd: every data in v_used be the value of element(input)
//Makes every element of this object be the value of the integer-valued (int-typed) operand.
```

인자값은 int 형(정수)이여야한다 (precnd) 넘겨받은 int 값으로 해당 벡터의 모든 data 들을 변환한다.

```
void MyIntVector::operator()(int x)
{
    for (int i = 0; i < v_used; i++)
        this->data[i] = x;
}
```

형 값을 인자로 받은 data 의 모든 값을 인자로 바꾼다. 바꿀 데이터의 인덱스 범위는 0 에서부터 크기 전까지이다.

Result:

```
cout << ">>Test ( ) Operator(Unary)" << endl << endl;
cout << "v1(5)" << endl;
v1(5);
printElem("v1", v1);
cout << "----------------------" << endl;
```

```
----------------------
>>Test ( ) Operator(Unary)

v1(5)
v1[0] : 5
v1[1] : 5
v1[2] : 5

----------------------
```

기존 1, 2, 3 의 원소값을 가진 v1 벡터가 () 연산자로 인해 모든 data 가 5 로 변하였음을 확인했다.

Function Pop_back : void pop_back();

```
void pop_back(); //pop back func
//precnd: func should work when v_used is more than 0
//postcnd: remove data at end of the vector(last used index)
//Removes the last element in the vector, effectively reducing the vector size by one and invalidating all references to it.
```

Pop_back 하고자 하는 대상이 있어야함으로 v_used 크기가 0 이면 안된다.(적어도 하나의 data 가 존재해야됨) (precnd) 수행되면 맨 마지막 원소를 제거한다.

```
void MyIntVector::pop_back()
{
    if(!(v_used > 0))
        throw std::out_of_range("nothing to pop back in this vector");

    data[--v_used] = NULL;
}
```

pop_back 함수는 맨 마지막 위치(크기 전)에 있는 값을 없애는 함수이다. 딱히 값을 바꿀 필요 없이 인덱스 연산자에 의해 범위가 제한되기 때문에 크기를 줄이기만 하면 된다.

Result:

```
cout << ">>Pop back v1 x 3" << endl << endl;

v1.pop_back( );
v1.pop_back( );
v1.pop_back( );
printVector("v1", v1);
cout << "----------------------" << endl;
```

```
>>Pop back v1 x 3


MyIntVector v1
capacity : 50
size : 3
```

앞서 v1+=v2 연산자로 1,2,3,1,2,3 의 data 를 가진 v1 에서 pop_back 을 3 번 사용하여 원래 길이인 1,2,3 으로 돌아오는 것을 출력한 결과물이다.

Function Push_back: void push_back(int x);

```
void push_back(int x); // puch back func
//precnd: factor should be integer
//postcnd: if v_capacity is full , double current capacity using reserve func
//Adds a new element at the end of the vector, after its current last element. The content of this new element is initialized to a copy of x.
```

삽입되는 인자(int x) 는 정수형이여야 한다. (precnd) 만약 총 용량 한계점에 도달하여 삽입되지 못 할경우 크기를 2 배로 늘려서 해결한다.

```
void MyIntVector::push_back(int x)
{
    if (v_used == v_capacity)
        reserve(v_capacity * 2);

    data[v_used++] = x;
}
```

push_back 함수는 맨 마지막 위치(크기)에 인자로 받은 int 형 정수 데이터 값을 추가한다. 만약 데이터를 넣기 전, 용량과 크기가 같을 경우 데이터를 더 이상 추가할 수 없기에 reserve() 함수를 이용하여 용량을 늘린다. 여기서는 원래 용량의 두 배를 늘리기로 결정했다. 데이터를 추가한 뒤, 크기 또한 1 증가시킨다.(v_used)

Result:

```
cout << ">>push back(1),(2),(3) into v2" << endl << endl;
v2.push_back(1);
v2.push_back(2);
v2.push_back(3);
printVector("v2", v2);
```

```
--------------------
>>push back(1),(2),(3) into v2


MyIntVector v2
capacity : 50
size : 3

v2[0] : 1
v2[1] : 2
v2[2] : 3
```

초기 default 생성자 예시의 빈 벡터 v2 에 push_back 을 차례로 이용하여 1, 2, 3,을 차례로 삽입하는 결과물이다.

Function capacity : size_t capacity() const;

```
size_t capacity( ) const; // capcity getter func
//precnd: None
//postcnd: Returns the size of the allocated storage space for the elements of the vector container.
```

```
size_t MyIntVector::capacity( ) const
{
    return v_capacity;
}
```

Function size: size_t size() const;

```
size_t size( ) const; // used size getter func
//precnd: None
//postcnd: Returns the number of elements in the vector container.
```

```
size_t MyIntVector::size( ) const
{
    return v_used;
}
```

각각 v_capacity 와 v_used 를 반환하는 getter 함수이다. Const 를 사용하여 변수변동이 일어나지 않도록 설정하였다.

Result:

```
void printVector(string name ,MyIntVector& v) {
    cout << endl;
    cout << "MyIntVector " << name << endl;
    cout << "capacity : " << v.capacity( ) << endl;
    cout << "size : " << v.size( ) << endl << endl;
}
```

앞선 예제들에서 사용했던 printVector 함수에서 두 getter 함수가 계속 사용되었다.

```
>>Test Default constructor

MyIntVector v1;

MyIntVector v1
capacity : 100
size : 0

MyIntVector v2(50);

MyIntVector v2
capacity : 50
size : 0
```

Function reserve : void reserve(size_t n);

```
void reserve(size_t n); // turn capacity into 'n'
//precnd: n(integer) should be more than 0
//postcnd: Requests that the capacity of the allocated storage space for the elements of the vector container be at least enough to hold n elements.
// if n is lower(or same) than size(v_used) fix n into v_used,
```

인자로 받는 n 은 0 보다 큰 정수형이 여야한다. (precnd) 현재 객체의 capacity 값을 인자로 받은 n 으로 바꾸고 v_used 값 또한 변경한다.

```
void MyIntVector::reserve(size_t n)
{
    int* temp;
    if (n <= v_used)
        n = v_used;

    temp = new int[n];
    std::copy(data, data + v_used, temp);
    delete[] data;
    data = temp;
    v_capacity = n;
}
```

용량의 크기를 바꾸며 data 또한 바뀐 용량으로 새로 할당해주는 함수이다. 인자로 들어온 n 에 값에 따라 바꾸는 용량이 결정이 되는데 만약 n 이 현재 크기보다 작거나 같을 경우 n 의 값을 크기로 바꿔 데이터가 손실되는 경우를 방지한다.

Result:

```
cout << ">>Test reserve(size_t n)" << endl << endl;
cout << "v1.reserve(2)" << endl;
v1.reserve(2);
printVector("v1", v1);
cout << "v1.reserve(20)" << endl;
v1.reserve(20);
printVector("v1", v1);
```

```
----------------------
>>Test reserve(size_t n)

v1.reserve(2)

MyIntVector v1
capacity : 3
size : 3

v1.reserve(20)

MyIntVector v1
capacity : 20
size : 3
```

처음 MyIntVector v1 으로 생성되었던 (capacity 100) v1 을 reserve 함수를 사용하여 각각 2, 20 을 인자로 넘겨주어 결과값을 출력하였다. 기존 v_used 크기보다 작은 값인 2 의경우 손실 방지를 위해 3 으로 값이 변경되어 적용됨을 확인 할 수 있다.

Function is_empty : bool is_empty() const;

```
bool is_empty( ) const; //Returns whether the vector container is empty,
                        //i.e., whether or not its size is 0.
//precnd: None
//postcnd: if v_used is 0 return true , v_used is not 0 return false
```

```
bool MyIntVector::is_empty( ) const
{
    if (v_used == 0)
        return true;
    else
        return false;
}
```

Function clear : void clear();

```
void clear( ); // All the elements of the vector are dropped: they are removed from the vector container,
               //leaving the container with a size of 0 and a default capacity.
//precnd: None
//postcnd: delete currnet data, pointer at NULL, initialize v_used , v_capacity into 0
```

```
void MyIntVector::clear( )
{
    delete[] data;
    data = NULL;
    v_used = 0;
    v_capacity = 0;
}
```

empty 함수는 data 배열에 아무런 값도 없을 경우 true 를 반환하고 아니면 false 를 반환하는 함수이다. 즉 크기가 0 인 경우와 아닌 경우를 따져주는 함수이다.

clear 함수는 멤버 변수를 초기화시키는 함수이다. 말 그대로 초기화시키기 때문에 data 가 가리킨 힙 메모리를 delete 하여 할당을 해제하고 다른 주소를 가리키지 않게 NULL 을 가리키게 한다. 사용하지 않기 때문에 크기와 용량 또한 0 으로 값을 바꾼다.

Result:

```
cout << ">>Test bool is_empty & void clear" << endl << endl;
cout << "v1.is_empty( ) : " << v1.is_empty( ) << endl;
cout << "v1.clear( )" <<  endl;
v1.clear( );
cout << "v1.is_empty( ) : " << v1.is_empty( ) << endl;

cout << "----------------------" << endl;
```

```
>>Test bool is_empty & void clear

v1.is_empty() : 0
v1.clear()
v1.is_empty() : 1
----------------------
```

지금까지 사용했던 v1 백터를 is _empty 로 점검한 후, clear 함수를 사용하여 data 를 삭제, 다시 is_empty 로 내부값을 점검하는 결과를 출력화면으로 알 수 있다.

과제 후기 & 고찰

벡터의 기본적인 기능들을 직접 구현해보는 과제였다. 코드 한단계씩 컨테이너가 어떻게 구성되고 작동되는지 학습하고 , 실습해 볼 수 있어서 유익한 시간이 되었다.

보고서의 작성에 시간이 오래걸려 soft deadline 에 맞추지 못한 것이 아쉽다.

실행 환경:

Windows 10 OS + Visual studio 2019 (Community 16.7.5ver)