OpenCV 프로그래밍을 위한 모던 C++ 정리

『OpenCV 4로 배우는 컴퓨터 비전과 머신 러닝』 저자

황 선 규

목차

- 연산자 오버로딩 (Operator overloading)
- std::vector 클래스
- 범위 기반 for (Range-based for)
- std::sort() 함수
- 람다 표현식 (Lambda expression)
- 스마트 포인터 (Smart Pointer)

연산자 오버로딩

- 연산자 오버로딩 (operator overloading)
  - C++ 기본 연산자를 사용자 정의 타입에서 새롭게 정의하여 사용

```
+ - * / % ^ & | ~ ! = < > ++ -- += -+ << >> -> () [] etc
```

```
int a = 10;
int b = 20;
int c = a + b; // 30
```

```
Point a(10, 20);
Point b(30, 40);
Point c = a + b;   // [40, 60]
```

  - 보통 사용자 정의 클래스를 설계할 때, 직관적인 코드 작성을 위하여 연산자 오버로딩을 구현함

- 연산자 오버로딩 구현 함수 형식

```
return_type operator op ( arguments );
```

실습: operator

연산자 오버로딩: 구현

☐ 클래스 멤버 함수로 구현

```
class Point {
public:
    int x, y;

    Point(0) : x(0), y(0) {}
    Point(int _x, int _y) : x(_x), y(_y) {}

    Point operator + (const Point& p) {
        return Point(x + p.x, y + p.y);
    }
};

int main()
{
    Point a(10, 20);
    Point b(30, 40);
    Point c = a + b;   // [40, 60]
}
```

연산자 오버로딩: 구현

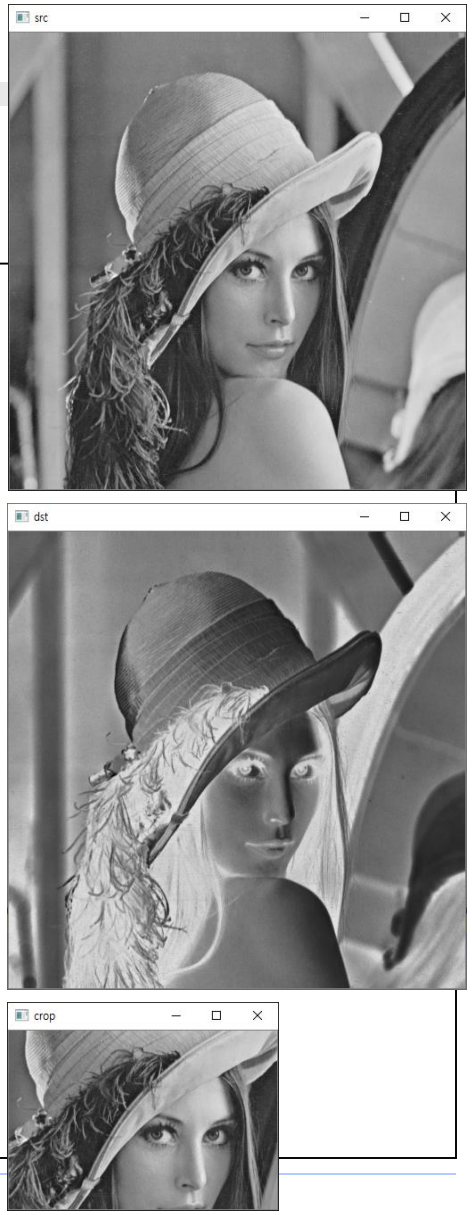
☐ 전역 함수로 구현

```
class Point {
public:
    int x, y;

    Point(0) : x(0), y(0) {}
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
};

inline Point operator + (int n, const Point& p)
{
    return Point(n + p.x, n + p.y);
}

int main()
{
    Point a(10, 20);
    Point b = 10 + a; // [20, 30]
}
```

연산자 오버로딩: 예제 코드

- OpenCV 연산자 오버로딩 사용 예제

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main(void)
{
    Mat src = imread("lenna.bmp", IMREAD_GRAYSCALE);
    Mat dst = 255 - src;

    Rect rc(100, 150, 300, 200);
    Mat crop = src(rc);

    imshow("src", src);
    imshow("dst", dst);
    imshow("crop", crop);

    waitKey(0);
    return 0;
}
```



std::vector 클래스

- vector란?
  - C++ STL에서 지원하는 대표적인 시퀀스 컨테이너 / 클래스 템플릿
  - 같은 타입의 데이터를 여러 개(가변 개수) 저장 가능
    - 특정 위치의 데이터를 마치 배열처럼([] 연산자 이용) 접근 가능
    - 새로운 데이터를 추가하거나 삭제 가능

- vector를 사용하려면?

```
std::vector<typename> var_name;
```

  - vector는 C++ template 문법으로 만들어진 클래스이므로 변수 선언 시 typename을 지정해야 함
  - <vector> 헤더 파일 포함 필요 (#include <vector>)
  - std 네임스페이스 사용 (using std::vector;)

std::vector 클래스

□ 주요 vector 사용법

```
vector<T> v1;              // T 타입을 저장할 벡터 v1 생성. v1은 현재 비어 있음.
vector<T> v2(v1);          // v1을 복사하여 v2를 생성

vector<T> v3(n);           // T 타입 객체 n개를 저장할 v3을 생성
vector<T> v4(n, val);      // T 타입 객체 n개를 저장하고 val로 초기화

vector<T> v5{ a, b, c, ... };     // {}안의 원소로 초기화
vector<T> v6 = { a, b, c, ... };  // v5와 동일

v[n];                      // n번째 원소에 접근. v.at(n); 과 같음.

v.empty();                 // v가 비어 있으면 true를 반환
v.size();                  // v에 포함되어 있는 원소 개수를 반환
v.capacity();              // v의 허용 용량을 반환

v.push_back(a);            // v 맨 뒤에 새로운 원소 a를 추가

v1 = v2;                   // v2를 복사하여 v1에 대입 (깊은 복사)
v1 = {a, b, c, ...};       // v1을 {} 괄호 안 데이터로 교체
```

std::vector 클래스: 예제 코드

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> func()
{
    vector<int> v {-10, -20};
    return v;
}

int main()
{
    vector<int> a;
    a.push_back(10);
    a.push_back(20);
    a.push_back(30);
    a.push_back(40);

    for (int i = 0; i < a.size(); i++)
        cout << a[i] << endl;

    vector<string> b {"I", "love", "you"};
    b.push_back("!!!");

    for (int i = 0; i < b.size(); i++)
        cout << b[i] << endl;

    vector<int> c(a.begin(), a.begin() + 3);

    for (int i = 0; i < c.size(); i++)
        cout << c[i] << endl;

    a.clear();
    cout << a.size() << endl;

    vector<int> d = func();

    for (int i = 0; i < d.size(); i++)
        cout << d[i] << endl;

    return 0;
}
```

범위 기반 for

- 범위 기반 for (Range-based for)란?
  - C++11에서 새로 지원하는 새로운 반복문 문법
  - 배열과 vector 등의 컨테이너의 원소를 순차적으로 접근
  - 장점: 코드가 간결해지고 직관적이다.
  - 단점: 현재 몇 번째 원소를 다루는 지 알 수 없다(?) (반복자도 동일)

- 범위 기반 for 루프 형식

```
for ( declaration : expression )
    loop-statement
```

범위 기반 for: 예제 코드

실습: rangedfor

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
  vector<int> ages {20, 30, 40, 50};

  for (int i = 0; i < ages.size(); i++) {
    int age = ages[i];
    cout << age << endl;
  }

  for (int age : ages)
    cout << age << endl;

  for (int& age : ages)
    cout << age << endl;

  for (auto& age : ages)
    cout << age << endl;

  string str[] = {"I", "love", "you"};

  for (auto& s : str)
    cout << s << endl;

  return 0;
}
```

std::sort() 함수

C 언어에서 정렬

```
#include <stdlib.h>

void qsort(void* base, size_t num, size_t size,
           int(*compar)(const void*, const void*));
```

C++ 언어에서 정렬

```
#include <algorithm>

template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

- qsort() 보다 빠르고, 배열 뿐만 아니라 벡터의 정렬도 지원

std::sort() 함수: 예제 코드

☐ 기본 사용법 (배열과 벡터의 정렬)

```
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    int num[5] = {2, 4, 1, 5, 3};

    sort(num, num + 5);                         // 오름 차순
//  sort(num, num + 5, greater<int>());         // 내림 차순
//  sort(begin(num), end(num));                 // std::begin(), std::end()

    vector<string> fruits = {"orange", "banana", "apple", "pear", "lemon"};

    sort(vec.begin(), vec.end());                           // 오름 차순
//  sort(vec.begin(), vec.end(), greater<string>());        // 내림 차순

    return 0;
}
```

sort() 함수: 예제 코드

☐ 클래스 객체의 정렬: < 연산자 오버로딩

```
class Person {
public:
    string name;
    int age;

    Person(string _name, int _age)
        : name(_name), age(_age) {}

    bool operator <(const Person &a) const {
        return this->age < a.age;
    }

    void print() {
        cout << name << ", " << age << endl;
    }
};
```

```
int main()
{
    vector<Person> v;
    v.push_back(Person("Emma", 20));
    v.push_back(Person("Ryan", 28));
    v.push_back(Person("Jane", 25));
    v.push_back(Person("Sam", 40));
    v.push_back(Person("William", 35));

    sort(v.begin(), v.end());

    for (Person p : v)
        p.print();

    return 0;
}
```

람다 표현식

- 람다 표현식 (Lambda expression)
  - C++11에서 새로 지원하는 이름 없는 함수 객체
  - 함수의 포인터 또는 함수 객체(function object)를 대체

- 람다 표현식 형식:

```
[captures] (params) -> ret { body }
```

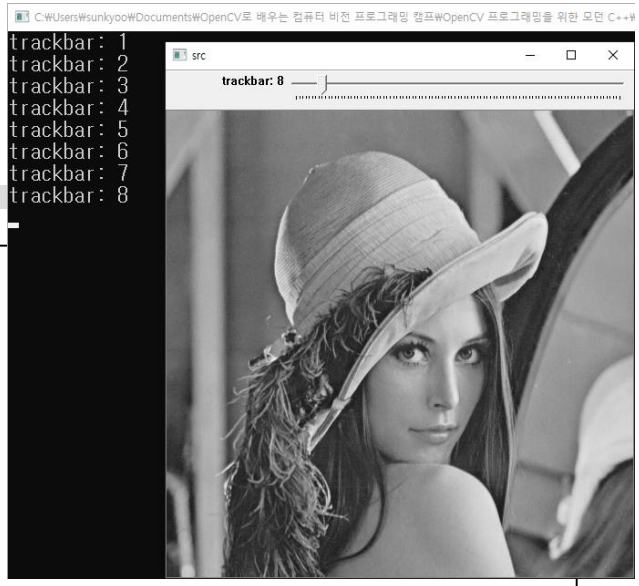
  - captures: 람다 표현식 외부 변수 접근 방법 지정
    - [=] : by-copy capture default
    - [&] : by-reference capture default
    - [] : Empty capture
  - params: 인자가 없으면 () 생략 가능
  - ret: 반환형이 명확할 경우 -> ret 생략 가능

실습: lambda

람다 표현식: 예제 코드

```
int main(void)
{
    [] {};      // 가장 간단한 형태의 람다 표현식
    []() {};    // 비어있는 인자
    [] {}();    // 호출까지 수행

    [](double a) { cout << a * a << endl; } (3.0);

    vector<Person> v;
    v.push_back(Person("Emma", 20));
    v.push_back(Person("Ryan", 28));
    v.push_back(Person("Jane", 25));
    v.push_back(Person("Sam", 40));
    v.push_back(Person("William", 35));

    sort(v.begin(), v.end(), [](const Person& a, const Person& b) {
        return a.name < b.name;
        });

    for (Person& p : v)
        p.print();
}
```

람다 표현식: 예제 코드




```
#include "opencv2/opencv.hpp"
#include <iostream>

using namespace cv;
using namespace std;

int main(void)
{
     Mat src = imread("lenna.bmp", IMREAD_GRAYSCALE);

     imshow("src", src);

     createTrackbar("trackbar", "src", 0, 100, [](int p, void*) {
        cout << "trackbar: " << p << endl;
        });

     waitKey(0);
     return 0;
}
```

스마트 포인터

- 스마트 포인터(smart pointer)란?
  - (동적 생성된 객체를 가리키는) 포인터 자료형을 흉내내어 만든 클래스로서, `->` 와 `*` 연산자를 오버로딩하여 포인터같은 역할을 수행
  - 동적 생성된 객체를 자동으로 해제하는 기능 제공
  - 다양한 타입을 지원하기 위해 클래스 템플릿으로 정의

- C++11에서 제공하는 스마트 포인터 클래스
  - std::shared_ptr, std::unique_ptr, std::weak_ptr, etc

- OpenCV에서 사용하는 스마트 포인터 클래스
  - cv::Ptr

```
template <typename _Tp>
using Ptr = std::shared_ptr<_Tp>;
```

실습: smartptr

스마트 포인터: 구현

☐ 스마트 포인터 기본 구현

```
#include <iostream>
#include <memory>
using namespace std;

class Rect {
public:
    void draw() {
        cout << "Rect draw!" << endl;
    }
};

class Ptr {
    Rect* obj;
public:
    Ptr(Rect* p = 0) : obj(p) {}
    ~Ptr() { delete obj; }
    Rect* operator->() { return obj; }
    Rect& operator*() { return *obj; }
};
```

```
int main()
{
    Ptr p = new Rect;
    p->draw();

    return 0;
}
```

스마트 포인터: 구현

- 스마트 포인터 기본 구현 (Con't)
  - 다양한 타입을 지원 → 클래스 템플릿으로 구현
  - 얕은 복사로 인한 이중 삭제 → 참조 계수 사용

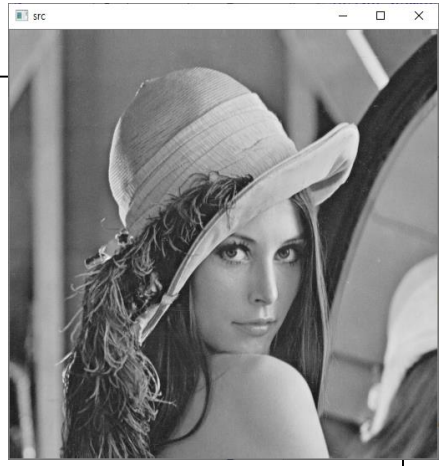
```
class Rect {
public:
    void draw() {
        cout << "Rect draw!" << endl;
    }
};

template<typename T> class Ptr {
    T* obj;
public:
    Ptr(T* p = 0) : obj(p) {}
    ~Ptr() { delete obj; }
    T* operator->() { return obj; }
    T& operator*() { return *obj; }
};
```

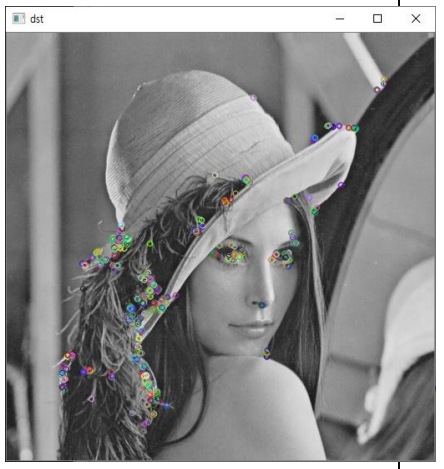
```
int main()
{
    Ptr<Rect> p = new Rect;
    p->draw();

    //Ptr<Rect> p2 = p;

    return 0;
}
```

스마트 포인터: 예제 코드

☐ OpenCV 스마트 포인터 사용 예제

```
int main()
{
    Mat src = imread("lenna.bmp", IMREAD_GRAYSCALE);

    Ptr<Feature2D> detector = ORB::create();

    vector<KeyPoint> keypoints;
    detector->detect(src, keypoints);

    Mat dst;
    drawKeypoints(src, keypoints, dst);

    imshow("src", src);
    imshow("dst", dst);

    waitKey(0);
    return 0;
}
```





break