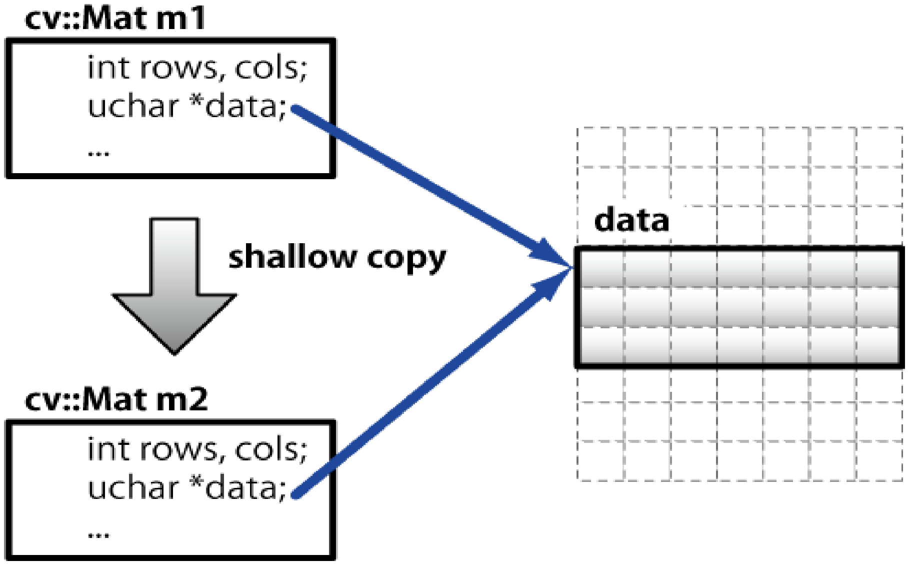
Memory Management/ Pixel Access

Sung Soo Hwang

Memory management

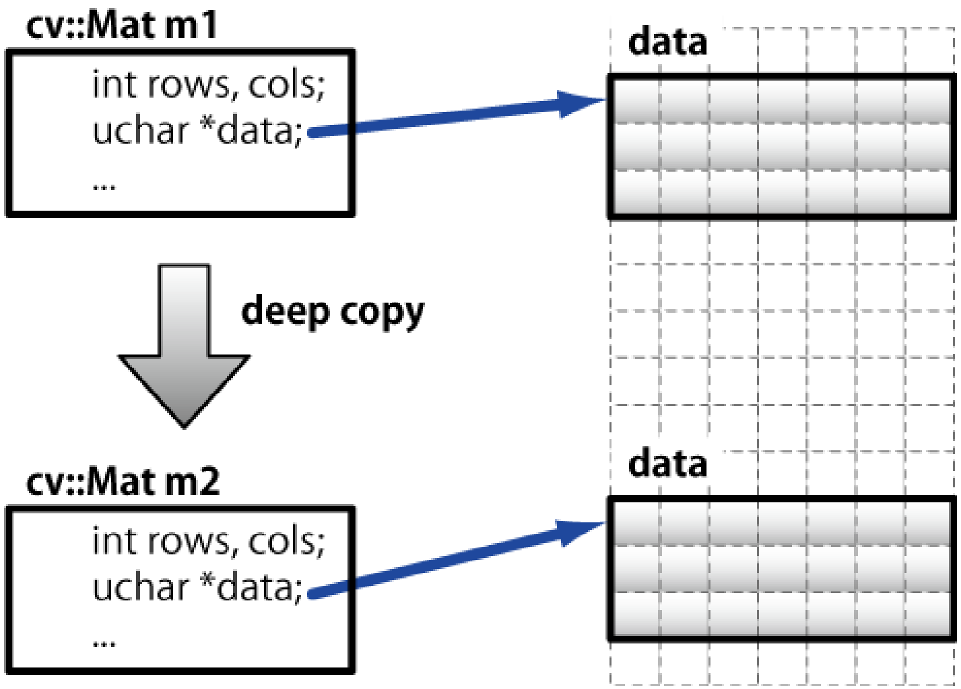
- Shallow copy
  - Mat data structure consists of header and data
  - In case of shallow copy, the address for data is copied
  - Use = for shallow copy
  - How about copyTo?

➔ when the destination matrix and the source matrix have the same type and size, copyTo will not change the address of the destination matrix



Memory management

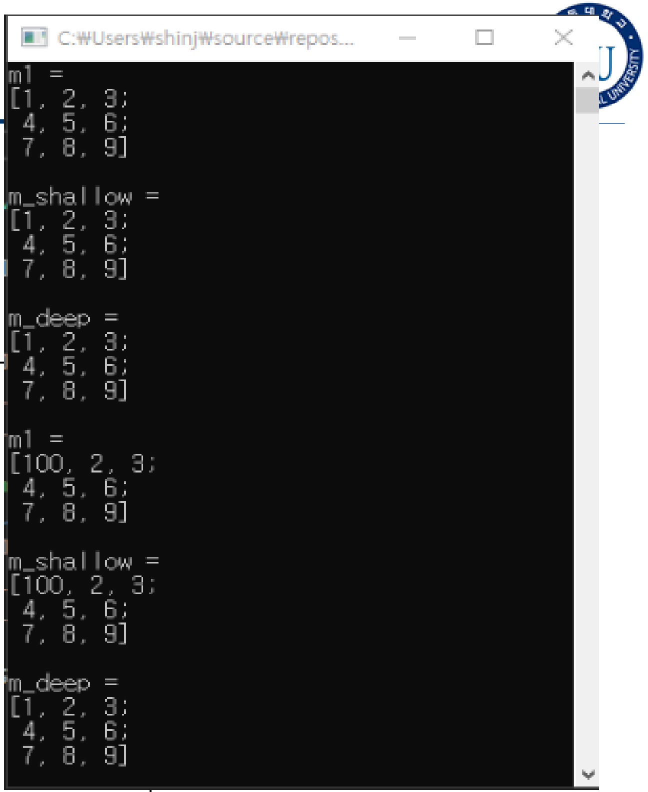
- Deep copy
  - Use clone() for deep copy
  - Mat creation and copyTo() are performed inside clone()



Memory management

- Shallow/Deep copy
  - Example code

```
int main() {
      Mat m1 = (Mat_ < double >(3, 3)
            << 1, 2, 3, 4, 5, 6, 7, 8, 9);

      Mat m_shallow = m1;
      Mat m_deep = m1.clone();

      cout << "m1 =₩n" << m1 << endl << endl;
      cout << "m_shallow =₩n" << m_shallow << endl << endl;
      cout << "m_deep =₩n" << m_deep << endl << endl;

      // Update m1
      m1.at < double >(0, 0) = 100;
      cout << "m1 =₩n" << m1 << endl << endl;
      cout << "m_shallow =₩n" << m_shallow << endl << endl;
      cout << "m_deep =₩n" << m_deep << endl << endl;

      waitKey(0);
}
```



```
m1 =
[1, 2, 3;
 4, 5, 6;
 7, 8, 9]

m_shallow =
[1, 2, 3;
 4, 5, 6;
 7, 8, 9]

m_deep =
[1, 2, 3;
 4, 5, 6;
 7, 8, 9]

m1 =
[100, 2, 3;
 4, 5, 6;
 7, 8, 9]

m_shallow =
[100, 2, 3;
 4, 5, 6;
 7, 8, 9]

m_deep =
[1, 2, 3;
 4, 5, 6;
 7, 8, 9]
```

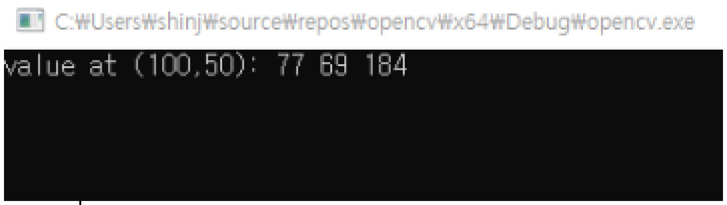
Pixel access

- By using at operator
  - image.at<DATA_TYPE>(WANT_ROW, WANT_COL)
    - DATA_TYPE: data type for a Mat (Ex: float, unsigned char)
    - WANT_ROW: the number of row to access
    - WANT_COL: the number column to access
  - Using at is a safe choice
    - It performs validity check
  - However, it is slow

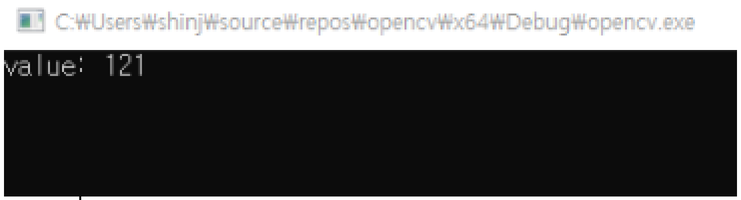
Pixel access

- By using at operator
  - Example code

```
int main() {
        Mat image, image_gray;
        int value, value_B, value_G, value_R, channels;

        image = imread("lena.png");
        image_gray = imread("lena.png", 0);
        //try both image & image_gray
        //channels = image_gray.channels();
        channels = image.channels();

        //At operator
        switch (channels) {
                case 1:
                value = image.at<uchar>(50, 100);
                cout << "value: " << value;
                break;

                case 3:
                value_B = image.at<Vec3b>(50, 100)[0];
                value_G = image.at<Vec3b>(50, 100)[1];
                value_R = image.at<Vec3b>(50, 100)[2];
                cout << "value at (100,50): " << value_B
                   << " " << value_G << " " << value_R << endl;
                break;
        }
        waitKey(0);
}
```



```
value at (100,50): 77 69 184
```

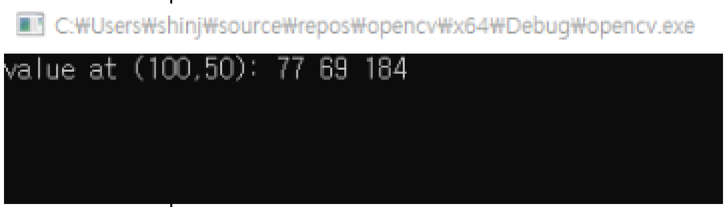


```
value: 121
```

Pixel access

- By using pointer
  - Faster than using at operator
  - Example code

```
int main() {
        Mat image = imread("lena.png");
        int value, value_B, value_G, value_R, channels;
        channels = image.channels();

        //Pointer
        uchar* p;
        p = image.ptr<uchar>(50);
        value_B = p[100 * channels + 0];
        value_G = p[100 * channels + 1];
        value_R = p[100 * channels + 2];

        cout << "value at (100,50): " << value_B << " "
        << value_G << " " << value_R << endl;

        waitKey(0);
}
```



```
value at (100,50): 77 69 184
```

Pixel access

- By using data member function
  - Fast
  - Hard to figure out inappropriate access

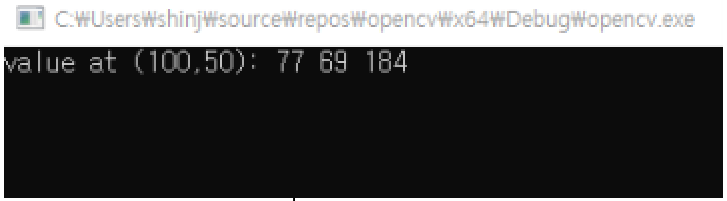
```
Mat image(ROW, COL, CV_TYPE);
DATA_TYPE* data = (DATA_TYPE*)image.data;
data[WANT_ROW * image.cols + WANT_COL]
```

- ROW : Number of Rows(Height)
- COL : Number of Columns(Width)
- CV_TYPE: Type type (ex: CV_8UC3 = 8 bit 3 channels)
- DATA_TYPE: Mat Date Type(Ex float, unsigned char)
- WANT_ROW: The row to access
- WANT_COL: The column to access

Pixel access

- By using data member function
  - Example code

```
int main() {
        Mat image;
        int value, value_B, value_G, value_R, channels;

        image = imread("lena.png");
        channels = image.channels();

        //Data member function
        uchar* data = (uchar*)image.data;
        value_B = data[(50 * image.cols + 100) * channels + 0];
        value_G = data[(50 * image.cols + 100) * channels + 1];
        value_R = data[(50 * image.cols + 100) * channels + 2];
        cout << "value at (100,50): " << value_B << " "
            << value_G << " " << value_R << endl;

        waitKey(0);
}
```



C:₩Users₩shinj₩source₩repos₩opencv₩x64₩Debug₩opencv.exe

```
value at (100,50): 77 69 184
```

Pixel access

- By using MatIterator
    - Example code

```
int main() {
    Mat image = imread("lena.png");
    Mat gray = imread("lena.png", 0);
    int value, value_B, value_G, value_R;
    // try both image & gray
    int channels = image.channels();
    MatIterator_ <uchar> it, end;
    MatIterator_<Vec3b> it3, end3;
    switch (channels) {
        case 1:
        for (it = image.begin<uchar>(), image.end<uchar>(); it != end; ++it) {
            value = *it;
            cout << "value: " << value << endl;
        }
        break;
        case 3:
        for (it3 = image.begin<Vec3b>(), end3 = image.end<Vec3b>(); it3 != end3; ++it3) {
            value_B = (*it3)[0];
            value_G = (*it3)[1];
            value_R = (*it3)[2];
            cout << "B: " << value_B << ", G: " << value_G << ", R: " << value_R << endl;
        }
        break;
    }
    waitKey(0);
}
```