CS205 C/C++ and Program Design – Project04

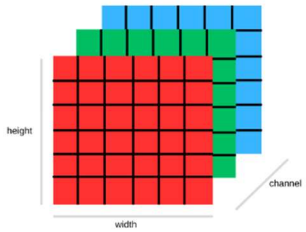
Fitria Zusni Farida (12112351)

Part 1 – Analysis

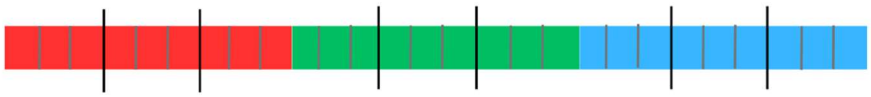
A class named Blob is created which consist of private members: height, width, channel, and data to represent the matrix data. The input matrix is 3D which has number of heights, width, and column but is represented using 1D array. A class template is used to support different data elements (can be unsigned char, short, int, float, double, etc).

```
private:
    int height ;
    int width ;
    int channel ;
    T* data ;
```

The 3D matrix is mapped into 1D matrix can be figured out as follow:



3D matrix



after mapped into 1D

To avoid linking error, the functions are defined in one header file under the class declaration. One main.cpp is used to test the function. The header file consists of several functions as follow:

1. Constructor
   - Default constructor

- Parameterized constructor
- Copy constructor

2. A Destructor
3. Function to access the private members: height, width, channel, data.
4. Operator overloading functions:
   - Arithmetic multiplication matrix (*)
   - Arithmetic addition matrix (+)
   - Arithmetic subtraction matrix (-)
   - Comparison operator (==)
   - Copy assignment (=)

Part 2 – Code

- Class Blob in header file op.hpp

```
template <typename T>
class Blob{

    private:
        int height ;
        int width ;
        int channel ;
        T* data ;

    public:

        // constructor
        Blob() ;
        Blob(int h, int w, int c, T* data) ;
        Blob(const Blob<T>& other);
        // destructor
        ~Blob() ;

        // get the members
        T* get_data() const;
        std::tuple<int,int,int> get_shape() const;
        int get_num_height() const;
        int get_num_width() const;
        int get_num_channel() const;

        //operator function
        Blob& operator=(const Blob<T>& other);
        bool operator==(const Blob<T>& other);
        Blob operator+(const Blob<T>& other);
        Blob operator-(const Blob<T>& other);
        Blob operator*(const Blob<T>& other);

} ;
```

- Function definition in header file op.hpp under the Blob class
  1. Constructor

```cpp
// consructor
template <typename T>
Blob<T>::Blob() {
    height = 0 ;
    width = 0 ;
    channel = 0 ;
    data = nullptr ;
}

template <typename T>
Blob<T>::Blob(int height, int width, int channel, T* data){
    this->height = height ;
    this->width = width ;
    this->channel = channel ;
    this->data = data ;
}

template <typename T>
Blob<T>::Blob(const Blob<T>& other){ //copy constructor

    // parameter checking
    if(&other == this){
        return ;
    }

    this->height = other.height;
    this->width = other.width;
    this->channel = other.channel;
    this->data = other.data;

}
```

Default constructor: will be automatically generated by the compiler if no arguments are passed. The member variables height, width, and channel will be set to zero and the data array will be nullptr.
Parameterized constructor: initialized the member variable with specific value arguments passed.
Copy constructor: creates a new object by making a copy of an existing object. It is used to initialize a new object of the same class. The argument passed is an existing object. If the existing object passed is the object itself, it will return the object itself.

2. Destructor

```
template<typename T>
Blob<T>::~Blob(){
    delete[] data ;
}
```

A destructor to release dynamically allocated memory associated with 'data' members of the Blob class. The 'delete[]' statement deallocates an array that was allocated using new[] operator to prevent memory leaks. The destructor will be automatically called when an object of the class is destroyed.

3. Get the private members

```
template <typename T>
int Blob<T>::get_num_width() const{
    return width ;
}

template <typename T>
int Blob<T>::get_num_height() const{
    return height ;
}

template <typename T>
int Blob<T>::get_num_channel() const{
    return channel ;
}

template <typename T>
std::tuple<int, int, int> Blob<T>::get_shape() const{
    return std::make_tuple(height, width, channel) ;
}

template <typename T>
T* Blob<T>::get_data() const {
    return data;
}
```

Functions to access the private member variable height, width, channel, and the data array in case we need to access them since it's in private. get_shape function: this function return the size of the mapped 3D matrix. std::make_tuple is a template function provided by the C++ Standard Library that creates a std::tuple object containing the provided arguments.

4. Operator overloading function
   - Operator overloading copy assignment =

```
template <typename T>
Blob<T>& Blob<T>::operator=(const Blob<T>& other){ // copy assignment

    if(this == &other){
        return *this;
    }

    height = other.height;
    width = other.width;
    channel = other.channel;

    if(data){
        delete[] data ;
    }


    data = other.data ;
    return *this ;

}
```

Copy assignment will replace the object with the another existing object passed as argument. If the argument passed is the object itself, the function will not assign the passed argument. Otherwise, the passed argument will be assigned and the original data array will be deallocated.

- Operator overloading ==

```
template <typename T>
bool Blob<T>::operator==(const Blob<T>& other){
    // parameter checking if nullptr
    if(data==nullptr || other.data==nullptr){
        throw std::invalid_argument("cannot compare since the data is null") ;
    }

    if(this->width != other.width || this->height != other.height || this->channel != other.channel || other.data == nullptr){
        return false ;
    }

    for(int i=0 ; i<width*height*channel ; i++){
        if(this->data[i] != other.data[i]){
            return false ;
        }
    }

    return true ;
}
```

This function is returning boolean true or false. If the height or width or channel of the two objects are different, it will necessarily return false. Otherwise, it requires to compare each of the elements of the data array. Parameter checking is created to check whether the array is nullptr since we will not compare anything if the data array is nullptr.

- Operator overloading +

```
template <typename T>
Blob<T> Blob<T>::operator+(const Blob<T>& other){
    //parameter checking size and if nullptr
    if(height != other.height || width != other.width || channel != other.channel){
        throw std::invalid_argument("Cannot do + operation since they have different size") ;
    }
    if(data == nullptr || other.data == nullptr){
        throw std::invalid_argument("Cannot do + operation since the data is null") ;
    }

    T * new_data = new T[height*width*channel];

    for(int i=0 ; i<width*height*channel ; i++){
        new_data[i] = data[i] + other.data[i] ;
    }

    return Blob<T>(height, width,channel,new_data) ;
}
```

Parameter checking is created to check if the two object has same matrix size and are not nullptr. The operation used by adding each element of the array data.

- Operator overloading –

```
template <typename T>
Blob<T> Blob<T>::operator-(const Blob<T>& other){

    //parameter checking size and if nullptr
    if(height != other.height || width != other.width || channel != other.channel){
        throw std::invalid_argument("Cannot do + operation since they have different size") ;
    }
    if(data == nullptr || other.data == nullptr){
        throw std::invalid_argument("Cannot do + operation since the data is null") ;
    }

    T* new_data = new T[height*width*channel] ;

    for(int i=0 ; i<height*width*channel ; i++){
        new_data[i] = data[i] - other.data[i] ;
    }

    return Blob<T>(height, width, channel, new_data) ;
}
```

Parameter checking is created to check if the two object has same matrix size and are not nullptr. The operation used by subtracting each element of the array data.

- Operator overloading *

```cpp
template <typename T>
Blob<T> Blob<T>::operator*(const Blob<T>& other){
    //parameter checking
    if(data==nullptr || other.data == nullptr){
        throw std::invalid_argument("Cannot do * operation since the data is null") ;
    }
    if(width != other.height || channel != other.channel){
        throw std::invalid_argument("Cannot do * operation since they have different size") ;
    }

    int len = height*other.width*channel ;

    T* new_data = new T[len]{0} ;

    for(int c=0 ; c<channel ; c++){
        for(int i=0 ; i<height ; i++){
            for(int j=0 ; j<other.width ; j++){
                int sum = 0 ;
                for(int k=0 ; k<width ; k++){

                    sum += (data[c*i*width + k] + other.data[c*k*other.width + j]);

                }
                new_data[i*other.width + j] = sum ;

            }

        }
    }

    return Blob<T>(height,other.width,channel,new_data);

}
```

Parameter checking is created to check if the two object has valid size of the matrix to do matrix operation and are not nullptr.

- Main function to check the functions work properly

Initialize the array to pass as argument

```cpp
int main() {

    int* dataInt2x3 = new int[18] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
    int* dataInt3x4 = new int[36] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,3
    float* dataFloat2x3 = new float[18] {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f,8.0f,9.0f,10.0f,11.0f,12.0f,13.0f,14.0f,15.0f,16.0f,17.0
    float* dataFloat3x4 = new float[36] {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f,8.0f,9.0f,10.0f,11.0f,12.0f,13.0f,14.0f,15.0f,16.0f,17.0
```

Call the constructor

```cpp
Blob<int> blobInt2x3(2,3,3,dataInt2x3) ;
Blob<int> blobInt3x4(3,4,3,dataInt3x4) ;
Blob<int> blobIntCopy(blobInt2x3) ;
Blob<float> blobFloat2x3(2,3,3,dataFloat2x3) ;
Blob<float> blobFloat3x4(3,4,3,dataFloat3x4) ;
```

Call the overloading operator function

```cpp
// do operation
//equals
Blob<int> intEquals2x3 = blobInt2x3 ;
Blob<float> floatEquals2x3 = blobFloat2x3 ;

//plus
Blob<int> plusInt2x3 = intEquals2x3 + blobInt2x3 ;
Blob<float> plusFloat2x3 = floatEquals2x3 + blobFloat2x3 ;

// minus
Blob<int> minusInt2x3 = intEquals2x3 - blobInt2x3 ;
Blob<float> minusFloat2x3 = floatEquals2x3 - blobFloat2x3 ;

// multiply
Blob<int> multiInt2x4 = blobInt2x3 * blobInt3x4 ;
Blob<float> multiFloat2x4 = blobFloat2x3 * blobFloat3x4 ;

// print the result ;
```

```
std::cout<<std::endl;
std::cout<< "Is blobInt2x3 is same with blobInt3x4? " ;
if(blobInt2x3==blobInt3x4){
    std::cout<<" YES."<<std::endl ;
} else {
    std::cout<<" NO." <<std::endl ;
}

std::cout<<std::endl;
std::cout<< "Is blobInt2x3 is same with intEquals2x3? " ;
if(blobInt2x3==intEquals2x3){
    std::cout<<" YES."<<std::endl ;
} else {
    std::cout<<" NO." <<std::endl ;
}
```

Part 3 – Result and Verification

By running the main function above, here is the output as expected:

```
PS D:\CS205\Project04> g++ main.cpp
PS D:\CS205\Project04> ./a.exe
Shape blobInt2x3x3: 2x3x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Shape blobInt3x4x3: 3x4x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
Shape blobFloat2x3x3: 2x3x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Shape blobFloat3x4x3: 3x4x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
Shape blobIntCopy: 2x3x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Shape intEq2x3: 2x3x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Shape floatEq2x3: 2x3x3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Shape intEquals2x3 + blobInt2x3 : 2x3x3
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
Shape floatEquals2x3 + blobFloat2x3: 2x3x3
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
Shape intEquals2x3 - blobInt2x3 : 2x3x3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Shape floatEquals2x3 - blobFloat2x3: 2x3x3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
shape blobInt2x3 * blobInt3x4 : 2x4x3
33 36 39 42 51 54 57 60 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
shape blobFloat2x3 * blobFloat3x4 2x4x3
33 36 39 42 51 54 57 60 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Is blobInt2x3 is same with blobInt3x4?  NO.

Is blobInt2x3 is same with intEquals2x3?  YES.
PS D:\CS205\Project04>
```

Part 4 – Difficulties and Solutions

1. Since the class template can handle various variable type passed for the data array, to make the testing and verification easier, the test displayed above is only testing int and float data type.
2. The object argument passed as parameter can only handle the same data type. For example, the copy constructor can only copy object with the same variable type, and the operator overloading function can only do operation of two object with different data type.