OBJECT ORIENTED PROGRAMMING

LABORATORY 4

OBJECTIVES

In this laboratory session, you will get familiarized with the *rule of three* from C++, so you will learn when and how to implement the destructor, copy constructor and assignment operator in C++.

THEORETICAL NOTIONS

Rule of three

The rule of three formalizes the rules on correct resource management and writing exception-safe code. It is related to three special member functions in a class: the copy constructor and the assignment operator and the destructor.

Both the copy constructor and the assignment operator are used when copying objects. The difference is that the copy constructor is used to initialize new objects as a copy of another object (as we are dealing with a constructor, the destination object is just being created), while the assignment operator works on existing objects (the destination object already exists). The destructor of an object is automatically called when the objects get out of scope (in the case of stack-allocated variables), or when delete is being called (in the case of heap-allocated variables).

The rule of three states that:

"If a class defines ***any*** *of the following special member functions (destructor, copy constructor or assignment operator) it should probably explicitly define* ***all three.****"*

If these special member functions are not explicitly implemented by the user, the compiler will automatically define them, as they are required in various situations (when passing function parameters by value, when returning objects from a function, when working with containers, just to name a few). The compiler-defined versions of these functions have the following semantics:

- Destructor – it just calls the destructors of all the class-type attributes of the current object;
- Copy constructor – builds all the attributes by calling the copy constructors of the object's class-type members. For non-class types (e.g., char, int, or pointer) data members it just uses assignment.
- Assignment operator – uses the assignment operator for all of the object's class-type members. For non-class types (e.g., char, int, or pointer) it just uses plain assignment.

Let's create an *Array* class that stores a heap-allocated array of characters and see these three functions in action:

```
#include <ostream>
#include <iostream>

using namespace std;

class Array
{
public:
   // precondition capacity > 0
   Array(unsigned int capacity = 10):m_data{new
int[capacity]},m_length{0},m_capacity{capacity}{}

   unsigned int lenght() const {return m_length;}
   unsigned int capacity() const {return m_capacity;}

   bool append(int elem){
       if(m_length < m_capacity){
           m_data[m_length++] = elem;
           return true;
       }
       return false;
   }

   // precondition: i < m_capacity
   int& at(unsigned int i){
       return m_data[i];
   }

   friend ostream& operator<<(ostream& os, const Array& a){
       for(unsigned int i {0}; i < a.m_length; i++)
           os<<a.m_data[i]<<" ";
       os<<endl;
       return os;
   }

private:
   int* m_data;
   unsigned int m_length;
   unsigned int m_capacity;
};


int main()
{
     Array a1{10};
```

```
    a1.append(1);
    a1.append(2);
    a1.append(3);
    a1.append(4);
    a1.append(5);

    cout<<"Array 1: "<<a1;
    Array a2{a1};
    cout<<"Array 2: "<<a2;

    Array a3{};

    a3 = a1;
    cout<<"Array 3: "<<a3;
    a2.at(0) = 10;

    cout<<"a2[0] = 10\nNow the arrays are:"<<endl;
    cout<<"Array 1: "<<a1;
    cout<<"Array 2: "<<a2;
    cout<<"Array 3: "<<a3;

    return 0;
}
```

This code will display:

```
Array 1: 1 2 3 4 5
Array 2: 1 2 3 4 5
Array 3: 1 2 3 4 5
a2[0] = 10
Now the arrays are:
Array 1: 10 2 3 4 5
Array 2: 10 2 3 4 5
Array 3: 10 2 3 4 5
```

which is not the expected result. When modifying array a2, we want to modify only the values stored in that array, and not also the values stored in a1 and a3.

The code has several problems:

- **memory management:** in the constructor of the array class we allocate memory on the heap, but we never release it. To fix this issue, we can implement the destructor of the class and release that memory:

```
~Array(){
      if(m_data){
          delete[] m_data;
          m_data = nullptr;
      }
  }
```

- **initializing an Array as a copy of another Array:** this is the responsibility of the copy constructor (in the code example: `Array a2{a1};`). The default copy constructor implemented by the compiler does a simple assignment between the m_data pointers of the newly constructed array and the array passed as a parameter. This is not correct, as in this case, the m_data field for both objects will point to the same memory location. To fix this issue, we need to explicitly implement the copy constructor of the class:

```
  Array(const Array& other){
      m_capacity = other.m_capacity;
      m_length = other.m_length;
      m_data = new int[m_capacity]();
      for(int i{0}; i < m_length; i++)
          m_data[i] = other.m_data[i];
  }
```

- **copying an Array into another (existing) Array:** this is the responsibility of the assignment operator (in the code example: `a3 = a1;`). The default assignment operator implemented by the compiler does a simple assignment between the m_data pointers of the two Array instances. This is not correct, as in this case, the m_data field for both objects will point to the same memory location. To fix this issue, we need to explicitly implement the assignment operator of the class. In the case of the assignment operator, because the destination object already existed, we need to explicitly free the memory to avoid memory leaks.

```
   Array& operator=(const Array& other){
       if(this != &other){ // self assignment check

           // release the memory
           if(m_data)
               delete[] m_data;

           // deep copy
           m_capacity = other.m_capacity;
           m_length = other.m_length;
           m_data = new int[m_capacity]();
           for(int i{0}; i < m_length; i++)
               m_data[i] = other.m_data[i];
       }
       // returning the current object
       // this is a pointer, so we need to dereference it
```

```
        return *this;
    }
```

Now the code works correctly, and we have no memory leaks;

The difference between the copy constructor and the assignment operator can be confusing at first, but is really not that difficult. You just need to keep in mind that the constructor is used to create new objects. Therefore:

- if a new object needs to be created before performing the copy, the copy constructor will be used. The copy constructor is also called when we pass a parameter by value, or when we return by value from a function.
- otherwise, if the "destination" object already exists and therefore the object doesn't need to be created, the assignment operator is used.

Below you have some examples of when the copy constructor and the assignment operators are called.

```
void f1(Array a){ // COPY CONSTRUCTOR CALLED: pass by value
     // your complex implementation here
}

Array f2(){
    // your complex implementation here
    Array a;
    return a; // COPY CONSTRUCTOR CALLED: return by value
}


int main(){
    Array a;

    Array c1 = a; // COPY CONSTRUCTOR CALLED
    Array c2{a}; // COPY CONSTRUCTOR CALLED
    Array c3 = Array{a}; // COPY CONSTRUCTOR CALLED

    c3 = c2; // ASSIGNMENT OPERATOR CALLED

   return 0;
}
```

PROPOSED PROBLEMS

1. Write a C++ class to implement a double-ended queue that stores integers. A double-ended queue, often abbreviated as deque, is a data structure that allows the insertion and deletion

of elements from both the front and the back. The data structure that you implement should be dynamic in size, meaning that it will dynamically adjust its size during runtime to accommodate a varying number of elements. The deque should be able to grow based on the number of elements it currently holds.

The deque should support the following operations:

- push_front(value): Add the given value to the front of the deque.
- pop_front(): Remove the element from the front of the deque.
- push_back(value): Add the given value to the back of the deque.
- pop_back(): Remove the element from the back of the deque.
- top(): Retrieve the element at the front of the deque without removing it.
- back(): Retrieve the element at the back of the deque without removing it.
- Overload << (stream insertion) and >> (stream extraction) operators: Implement these operators to allow easy input and output of deque elements.

Additionally, you **need** to implement the **Rule of Three** for proper memory management:

- Copy Constructor (Deque(const Deque& other)): creates a deep copy of the contents of another deque.
- Copy Assignment Operator (Deque& operator=(const Deque& other)): assigns the contents of another deque to the current deque.
- Destructor (~Deque()). Release any dynamically allocated memory.

Your implementation should ensure proper memory management! Write comprehensive test cases to catch all possible errors and ensure the correctness of your implementation. Test for scenarios such as: empty deque operations, adding and removing elements from both ends, resizing, deep copies, and proper memory cleanup.
Alternatively, if you find the deque data structure is too difficult, you can implement a dynamic queue data structure.

2. It is well known that the primitive data types have a limited range of values. For example, the range of values for a *long long int* is typically from -2^{63} to $2^{63}-1$. If we need to work with really really big numbers, we cannot use primitive data types.

a. Write a class called BigInteger to represent a large integer. You have two options: you either use a **heap allocated** array of digits and a variable to indicate the sign of the integer to store the number, or you use the *Deque* class from the previous exercise. (Don't change the name of the class, because the tests won't work! The class should be stored in)

b. Implement a default constructor and a parameterized constructor for your class. The default constructor initializes the number to 0. The parameterized constructor takes as input a string (std::string) of characters in the interval [0, 9] and stores the digits in corresponding fields.

c.

d. Write a function:
int compare(const BigInteger& N) const;

that compares the current object with the object N. The function will return -1 if the number stored in the current object is less than N, zero if the numbers are equal and 1 otherwise.

e. Use the compare function to overload the comparison operators ==, <, <=, >, and >=. **Don't duplicate the code.**

f. Ensure that the rule of three(copy constructor, assignment operator, and destructor for the BigInteger class) is properly implemented. If you use the *Deque* class do you still need to implement the rule of three for the BigInteger class?

g. Write the functions:
BigInteger add(const BigInteger& N) const;
BigInteger sub(const BigInteger& N) const;

and use them to overload the arithmetic operators: +, +=, -, -= (**reuse code, don't copy-paste the code**).
This design is intended to separate the task of performing the operation (like addition or comparison) from the task of overloading the operator. Each task presents its own challenges, and it is better to face them separately. **Implement the functions first, then overload the operators, not the other way around.**

When implementing these operations you might need several auxiliary functions to make your implementation cleaner:

1. a function used to prepend zeros to an existing array

You can use this function to align two numbers before processing them. For example, if you want to add the numbers 123 and 46572 before adding them, you should prepend to zeros to 123:

0	0	1	2	3+
4	6	5	7	2
4	6	6	9	5

2. a function to negate the values in an array

This function will multiply (element-wise) with -1 each element from the array and return the result. The original array is left unchanged! You could use this function when processing negative numbers or for the subtraction operation.

3. a function to add two arrays

This function takes as parameters two arrays of the same length (so you need the function at step 1), adds them (elementwise), and returns the result. You don't need to do anything if the result is greater or equal to 10 (if the result is not a digit), or if the result is smaller than 0. You will use the next function in such cases.

0	0	1	2	9+
4	6	5	8	2
4	6	6	10	11

For example, if you need to add a positive number with a negative number 46582 + (-729). You could use the negate function (the function written at step 2), and call the add method.

4	6	5	8	2+
0	0	-7	-2	-9
4	6	-2	6	-7

4. a function to "digitize" an array after the addition/subtraction operations
This function takes as input an array, and normalizes it such that each element is a digit (this function can be used after the function written at step 3).

When you have numbers that are greater than 9 in your array, you will need to use a *carry*. Initially, the carry will be 0.

You will start traversing the array from the last element, and to each element, you will add the previous *carry* value. If the result is larger than 10, you will subtract 10 from it and you will have a *carry* of 1.

Original	4	6	6	10	11
After digitize	4	6	7 (6 + 1)	1 (10 + 1 - 10)	1 (11 **- 10**)
	carry 0	carry 0	carry 0	**carry 1**	**carry 1**

When you have numbers that are negative in your array, you will need to use a *borrow*. Initially, the *borrow* will be 0.

You will start traversing the array from the last element, and to each element you will subtract the previous *borrow* value. If the result is larger than 10, you will add 10 from it and you will have a *borrow* of 1.

Original	4	6	-2	6	-7

After digitize	4	5 (6 -1)	8(-2 + 10)	5 (6 -1)	3 (-7 +10)
	borrow 0	borrow 0	**borrow 1**	borrow 0	**borrow 1**

With the help of these functions the BigInteger operations are trivial to implement. **Of course, you can come up (and you are encouraged to do so!) with your own idea of implementation.**

h. Write the function
std::string to_string(); which will represent the BigInteger into a string format.
Use this function when overloading the stream insertion operator <<.

i. Overload the post-increment (x++) and pre-increment operators (++x) for this class.

BigInteger& operator++(); // this is the pre-increment operator
BigInteger operator++(int); // this is the post-increment operator; the int parameter is a dummy parameter, meaning that you won't use it in the implementation, it is just used to distinguish between the pre-increment (++x) and post-increment (x++)

Both these operators should increment the big integer by 1 (**take into consideration the sign of the big integer when implementing this!**).
The pre-increment operator should return a reference to the current object **after** it is incremented.
The post-increment operator should return the value of the object **before** it was incremented. You can reuse the additional functions.

In the *main.cpp* file write the code that generates the first *N* Fibonacci numbers. You will see that you soon have an overflow even when using long long data type for storing the Fibonacci numbers. Modify the provided code in *main.cpp* such that it uses the *BigInteger* class.

Write your own tests (using assertions) to test the Fibonacci sequence.

EXTRA CREDIT

Image processing library

Image processing is a field in which various digital signal processing techniques are applied on images to enhance their quality or to extract some information from the images. For this project you will implement a basic image processing application.
The building block of the project will be a class that implements the image abstract data type. The application will support only grayscale images. A grayscale image is represented by a 2D array of pixels, and each pixel can take a value from the interval [0, 255], where the value 0 represents black, and the value 255 represents white. Values in between encode different shades of gray (the higher the value, the lighter the gray tone is).

The image and other helper classes

Implement a *Size* class that encapsulates the dimensions of an object. The class should contain as attributes two unsigned int for the width and height of the object. Implement a default constructor and a parameterized constructor, as well as getters and setters for the class attributes. Also implement the comparison operators for this class (==, <, <=, >, >=). Use the area of the objects to compare them.
Write a *Point* class to represent a point in a two-dimensional space; the class should have two integer attributes for the *x* and *y* coordinates of the point, getters and setters for these variables. Implement a default constructor which initializes a point to *(0, 0)*, and a parameterized constructor.
Implement a *Rectangle* class which encapsulates a rectangular area. The attributes of this class are *x, y* two integers which represent the coordinates of the top-left corner of the rectangle, and *width* and *height* the dimensions of the rectangle. Implement a default constructor, a parameterized constructor and getters and setters for all the class attributes. This class should also overload the addition and subtraction operators with a *Point*; these functions will be used to translate the rectangle. Overload the binary *and* & and *or* | operators. The & operator will compute the intersection between two rectangles, while the | operator will compute the union of two rectangles.

The declaration for the *Image* class is given below:

```
class Image
{
public:
    Image();
    Image(unsigned int w, unsigned int h);

    Image(const Image &other);
    Image& operator=(const Image &other);
    ~Image();
    bool load(std::string imagePath);
    bool save(std::string imagePath) const;

    Image operator+(const Image &i);
    Image operator-(const Image &i);
    Image operator*(double s);
    bool getROI(Image &roiImg, Rectangle roiRect);
    bool getROI(Image &roiImg, unsigned int x, unsigned int y, unsigned int 
width, unsigned int height);

    bool isEmpty() const;

    Size size() const;
    unsigned int width() const;
    unsigned int height() const;

    unsigned char& at(unsigned int x, unsigned int y);
    unsigned char& at(Point pt);

    unsigned char* row(int y);

    void release();

    friend std::ostream& operator<<(std::ostream& os, const Image& dt);
    friend std::istream& operator>>(std::istream& is, Image& dt);
    static Image zeros(unsigned int width, unsigned int height);
    static Image ones(unsigned int width, unsigned int height);

private:
    unsigned char** m_data;
    unsigned int m_width;
    unsigned int m_height;
};
```

The attributes of the *Image* class are:

- *m_data* - a pointer to a pointer of unsigned char, which will store the image pixels. This data must be dynamically allocated on the heap.
- *m_width* and *m_height* - two variables of type unsigned int, which store the width and height of the image.

The class provides two constructors: (1) the default constructor, in which the *m_data* pointer should be initialized to *nullptr*, and the width and the height of the image to zeros; (2) a parameterized constructor, which creates a new *Image* instance with the specified width and height; all the pixels in the created image should be initialized to zero.
As the *Image* class uses dynamically allocated memory, the *rule of three* must be implemented for this class: the copy constructor, the assignment operator and the destructor. The release method releases all the dynamically allocated memory in the object and should be called in the destructor of the class.
Implement getters and setters for the width and height of an image. In the *Image* class implement the method *Size size() const;* which returns the size (width and height of an image) as a *Size* object.

The *isEmpty* method returns true only if the image is empty (*m_data* pointer is *nullptr* and the width and height of the image are zero).

Images can be loaded or saved to the filesystem in the PGM (Portable Gray Map), due to its simple structure which makes it easy to understand and modify both manually and programmatically. PGM files have a simple ASCII structure and contain human-readable text, making them easy to interpret and edit.

Below is an example of a PGM image file:

```
P2
# Simple pgm image example
4 4
255
0 0 0 0
0 255 255 0
0 255 255 0
0 0 0 0
```

The file begins with the magic number *P2,* which indicates that this is an ASCII PGM file. The next line starts with the # character and contains a comment. The third line (*4 4*) specifies the width and height of the image, while the next line (255) stores the maximum pixel value. The following lines contain the actual image pixels.
In the *Image* class, implement the stream insertion and stream extraction operator to write and respectively write the image data (in the PGM) format to a stream. Next, implement the *load* and *save* method to load and save images on the filesystem using the PGM image format. These methods return a boolean value which indicates whether the operation was successful.
The *Image* class also contains two static factory methods, *zeros* and *ones*, which create and return an image of a specified size with all pixels set to zero (the *zeros* methods) or one (the *ones* method).
Several methods from the *Image* class offer pixel wise access. The *Image* class contains two overloaded methods *at*, which access or modify a pixel value at a specific location. One version takes x and y coordinates, and the other takes a Point object. The *row* method returns a pointer to the beginning of a specified row in the image data and can be used for row-wise processing.
Implement the + and - operators for the pixel wise addition and subtraction between two images. These operations only make sense for images of the same size; if the images have different resolutions, an exception will be thrown. Also, implement an multiplication operator for multiplying the image pixels by a scalar value (*operator**). This operator will take a scalar value and multiply each pixel in the image by this value.
A Region of Interest (ROI) is a term used in image processing and computer vision to refer to a specific area of an image that is selected for a particular operation or analysis. The ROI is defined by its spatial location within the image and, in this project, rectangular ROIs will be used. The methods *getROI* are overloaded methods to extract a sub-image or region of interest. One takes a *Rectangle* object to specify the ROI, and the other takes the top-left corner coordinates along with width and height.

<u>Image processing classes</u>

The foundational class for image processing operations is the *ImageProcessing* interface, which contains a pure virtual method with the following signature:

```
void process(const Image& src, Image& dst);
```

, where *src* represents the source image to be processed, and *dst* is the destination image that will store the result of the processing operation.
The subclasses of *ImageProcessing* class will override this method and implement a specific image processing operation.
Implement subclasses of the *ImageProcessing* interface for the following operations:

1. Brightness and contrast adjustment

This processing type operates at each pixel independently and alters the brightness and contrast of the image, as follows:

$$dst(x,\ y) = \alpha \cdot src(x,\ y) + \beta$$

, where src(x, y) and dst(x, y) represent the pixels at coordinates (x, y) from the input image and the filtered image respectively.

The parameters of this operation $\alpha>0$ (the gain) and β (the bias) can be considered to control the contrast and brightness of the image. These parameters will be specified in the constructor of the class. The default constructor initializes $\alpha = 1$ and $\beta = 0$ (so, in this case, this operation is a no-op).
The *process* method iterates through each image pixel and applies the transformation to each pixel. In the implementation, ensure that the result is computed on a real data type and that the final result is in the range [0, 255]. If not, the result should be clipped to this interval.

2. Gamma correction

Gamma correction is used to correct the overall brightness of an input image with a non-linear transformation function:

$$dst(x,\ y) = src(x,\ y)^{\gamma}$$

, where *dst*(x, y) and *src*(x, y) are the values of the pixels at coordinates (x, y) in the filtered image and input image, respectively, and γ is the gamma encoding factor. This operation is needed because humans perceive light in a non-linear manner; the human eye is more sensitive to changes in dark tones than to changes in lighter ones. So, gamma correction is used to optimize the usage of bytes when encoding an image, by taking into consideration this perception difference between the camera sensor and our eyes.

A value of γ less than 1 will darken the image, while a value larger than 1 will make the image appear brighter. If γ = 1 then the image will be left unchanged (no-op).
The value of γ should be passed as a parameter to the constructor of this class.

3. Image convolution

Convolutions are highly used in image processing and machine learning to extract some features from the input image. They involve the usage of a kernel (or filter) that is convolved over the input image as follows:

$$F = K * I$$

, where *F* is the output (filtered) image, *K* is the convolutional kernel and *I* is the input image. The convolutional kernel *K* has the shape (*w, h*), where *w* and *h* are usually odd numbers: *w = 2k + 1*.
At each image position (x, y), the convolutional filter is applied as follows:

$$F(x,\ y) = \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} K(u,\ v) I(x - u + k,\ y - v + k)$$

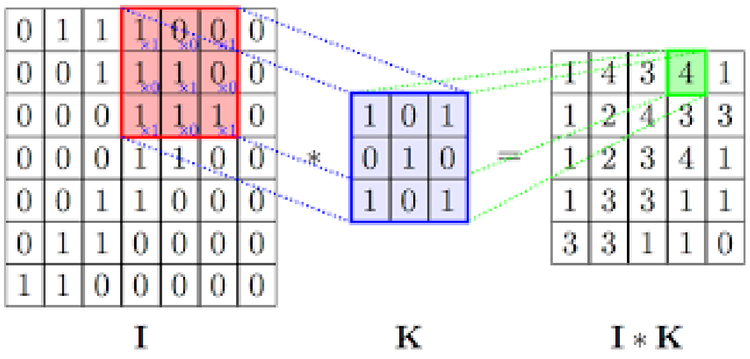

Figure P1. Illustration of image convolution.

The constructor of the *Convolution* class should also take a parameter a pointer to a function that is used to scale the result (of applying the convolution kernel) to the range [0, 255].

Some well-known convolutional kernels are:

0	0	0	**Identity kernel**. No-op. Applying convolution with this kernel results in the same image.
0	1	0	
0	0	0	
1	1	1	**Mean blur kernel**. (for this kernel, the scaling function should just **multiply** the convolution result to 1/9)
1	1	1	
1	1	1	

1	2	1	**3x3 Gaussian blur kernel**. Still a blurring kernel, but in this case, the central pixel of the kernel has a higher weight in the mean operation; for this kernel, the scaling function should just multiply the convolution result to 1/16.0)
2	4	2	
1	2	1	
1	2	1	**Horizontal Sobel kernel.** Used to detect horizontal edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
0	0	0	
-1	-2	-1	
-1	0	1	**Vertical Sobel kernel.** Used to detect vertical edges. In this case, the scaling function should be a linear mapping function that converts the range [-4*255, 4*255] to the range [0, 255]).
-2	0	2	
-1	0	1	

Drawing module

Create a module for drawing shapes over images. The module for drawing shapes over images aims to provide a simple and intuitive interface for adding basic geometric shapes such as circles, lines, and rectangles onto images. This can be useful for annotations, highlighting areas of interest, or creating simple graphics.

The module should contain at least the following functions:

- *void drawCircle(Image& img, Point center, int radius, unsigned char color);*
- *void drawLine(Image &img, Point p1, Point p2, unsigned char color);*
- *void drawRectangle(Image &img, Rectangle r, unsigned char color);*
- *void drawRectangle(Image &img, Point tl, Point br, unsigned char color);*

In practice more complex rasterization algorithms are used to convert 2D shapes into a rasterized format, but for this simple project it is sufficient to use the basic formulas for the shapes to determine the position of the points that lie on the shape's boundary.

EXTRA