## Operator overloading

In C++ the functionalities of the operators can be overloaded by defining `operator` functions, provided that:

- at least one of the arguments is a class instance;
- the operator is one of the predefined C++ operators.

The precedence, associativity, and number of arguments are fixed by the language and cannot be changed.

```
1 void operator <<
2     (stack<int> istack, int data)
3 {
4     istack.push(data);
5 }
```

Listing 1: Overload as non-member function

```
1 void Stack::operator <<
2     (int data)
3 {
4     push(data);
5 }
```

Listing 2: Overload as member function

**Exercise 1.** Using the STL class `stack`, overload the left bitshift operator `<<` to be used for stack push as a global function (see Listing 1).

**Exercise 2.** Modify the class `Stack` from Lab 09, overload the left bitshift operator `<<` to be used for stack push as a member function (see Listing 2).

*When overloading an operator as a member function of a class, the left operand is always the implicit parameter.*

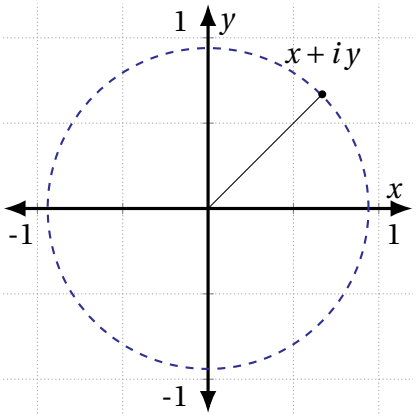
## Complex numbers

Complex numbers are numbers of the form $x + iy$, where $x, y \in \mathbb{R}$, and $i = \sqrt{-1}$ is the imaginary unit. We will implement a class `Complex` that represents complex numbers arithmetics.

**Exercise 3.** Write definition of the class `Complex` that represents complex numbers. The class has to `double` private data fields: one for the real, and one for the imaginary part. The class must have a default constructor, member function `set()`, and member function `print()`.

```
class Complex
{
public:
    Complex();
    void set(double real, double imag);
    Complex sum(const Complex& z);
    void print() const;
private:
    double real;
    double imag;
};
```

Listing 3: Definition of class `Complex`




**Exercise 4.** Implement the default constructor of class `Complex`, also `set()` and `print()`. Memeber functions. Extend the class with a parameter constructor as well.

```
void Complex::set
    (double real, double imag)
{
    this->real = real;
    this->imag = imag;
}
```

Listing 4: Set data fields

```
void Complex::print()
{
    cout << "(" << real;
    cout << ", " << imag;
    cout << ")";
}
```

Listing 5: Print complex number

**Exercise 5.** Let $(a+ib)$ and $(c+id)$ are two complex numbers. Their sum is defined:

$$(a+ib)+(c+id)=(a+c)+i(b+d). \quad (1)$$

Implement a member function `sum()` that calculates the complex sum the implicit parameter with another complex number, passed as explicit parameter of the function. Note that the result is a new object of type `Complex`.

```
Complex Complex::sum
            (const Complex &z)
{
    Complex result;
    result.real = real + z.real;
    result.imag = imag + z.imag;
    return result;
}
```

Listing 6: Sum two complex numbers

**Exercise 6.** Rework the function `sum()` from Exercise 6 by overloading the operator `+`. The header of the member function will be:

```
Complex Complex::operator +(const Complex &z)
```

**Exercise 7.** Let $(a+ib)$ and $(c+id)$ are two complex numbers. Then, their subtraction is:

$$(a+ib)-(c+id)=(a-c)+i(b-d). \quad (2)$$

Complex multiplication is:

$$(a+ib)(c+id) = (ac-bd)+i(ad+bc). \tag{3}$$

And complex division is:

$$\frac{a+ib}{c+id} = \frac{(ac+bd)+i(bc-ad)}{a^2+d^2}. \tag{4}$$

Overload operators `-`, `*` and `/` for class `Complex`.

## Overloading extraction and insertion operators

The bitwise right shift operator `>>` in the context of input streams is called *extraction operator*. The bitwise left shift operator `<<` in the context of output streams is called *insertion operator*. Both the can be overloaded, and then directly applied on objects of a given class. They must be overloaded as friend functions, because their first operand is a stream object.

*Extraction `>>` and insertion `<<` operators can be overloaded only as non-member functions.*

```
1 ostream& operator <<(ostream &out, const Complex &z)
2 {
3     cout << z.real << " + i." << z.imag;
4     return out;
5 }
```

Listing 7: Overloaded insertion operator

**Exercise 8.** Overload both extraction and insertion operators as friend functions of class `Complex`. After that `Complex` objects can be easily read and print from standard input and output:

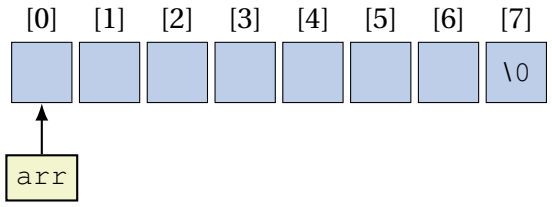
```
Complex z;
cin >> z;
cout << z;
```

## Automatic memory management

Class encapsulation hides dynamic memory management from the users of the class. The object variable is located in run-time in the stack, while the dynamic memory, allocated by it, is in the heap. The process in which the object allocates and frees the memory in the heap is referred to as *automatic memory management*. It is provided by the following class members: default constructor, copy constructor, destructor and overloaded assignment operator.

To illustrate automatic memory management, we will implement our own class `String`.

```
class String
{
public:
    String();
    String(const String& str);
    ~String();
    String& operator =(const String& str);
private:
    char* arr;
    int len;
};
```




Listing 8: Class `String` definition

**Exercise 9.** Write the definition of the class `String`. The private members are a pointer to character array, and an integer to store the length of the string. The string will have an extra character for zero terminator.

**Exercise 10. Default constructor** is invoked when an object is constructed with no arguments. It has to initialize the private data members.

Implement the default constructor of the class `String`. It initializes the pointer `arr` with `nullptr`, and the length `len` with `0`.

*If the pointers which will refer to dynamically allocated memory are not initialized, this can lead to errors which are difficult to track down.*

```
String::String(const String &str)
{
    len = str.len;
    arr = new char[len + 1];          // dynamic memory allocation
    for (int i = 0; i < len; i++)
    {
        arr[i] = str.arr[i];          // copy string contents
    }
    arr[len] = '\0';                  // terminating character
}
```

Listing 9: Copy constructor copies the contents of the parameter object

*When object allocates memory in the heap, provide copy constructor that will determine how a copy of the object is constructed.*

**Exercise 11. Copy constructor** is invoked whenever a copy of the object is needed, for example when the object is passed as a function parameter by value. The contents of the parameter object have to be copied in the object, created by the copy constructor.

Implement the copy constructor of the class `String`. It allocates memory for the dynamic array and copies the array of the parameter element by element.

```
String::~String()
{
    if (arr)                          // check if there is memory allocated
    {
        delete[] arr;                 // free the memory
        arr = nullptr;
    }
}
```

Listing 10: Destructor frees the memory, allocated by the object

**Exercise 12. Destructor** is a member that is automatically invoked when the object is destroyed. It is used to free any memory in the heap that has been allocated by the object.

Implement the destructor of the class `String`. Note that the pointer is verified to be different from `nullptr` before deleting it, and is set to `nullptr` after `delete`.

*Verifying pointers before deleting them, and setting them to `nullptr` after that, prevents memory management mistakes.*

```
String& String::operator =
            (const String &str)
{
    if (this != &str)
    {
        if (arr)
        {
            delete[] arr;
            arr = nullptr;
        }
```

Listing 11: Prevent self-copy and delete memory

```
        len = str.len;
        arr = new char[len + 1];
        for (int i = 0; i < len; i++)
        {
            arr[i] = str.arr[i];
        }
        arr[len] = '\0';
    }
    return *this;
}
```

Listing 12: Allocate memory and copy array

**Exercise 13. Assignment operator** copies the contents of an object into another. It takes care to free and alocate memory for the object when needed.

Implement the assignment operator of the class `String`.

## Problems

1. Overload the relational operator == for the class `Complex`. It must be a predicate function.
2. Overload selection and extraction operators for class `String`. Both functions must be global and friend of class `String`.