Overloading

- Function overloading
- Operator overloading

Function overloading

Overloading means that we reuse a function name with the following rules:

- Parameter list must be different
- Return type doesn't matter

We can overload normal functions as well as member functions

In [ ]:
```
double waterVolume(long long raindrops, double dropVol); // (1)
double waterVolume(double area, long long depth);        // (2)
```

How does the compiler do the name resolution?

In [ ]:
```
waterVolume(4934, 123.245); // resolves to (1)
waterVolume(1.23, 5563);    // resolves to (2)
waterVolume(45655, 23454);  // ERROR not defined
```

In [ ]:
```
waterVolume(45655.0, 23454); //resolves to (2)
waterVolume(45655, 23454ll); //resolves to (2)
```

Overloading operators

Operator overloading allows you to (re)define the behavior of operators for custom types. It enables you to use operators such as `+`, `-`, `*`, `/`, `==`, and others with user-defined objects, just like you would with built-in types. In this way you can make your code more intuitive, expressive, and natural to work with.

Let's see an example:

In [ ]:
```
class Point2D {
  private:
    double x_=0.0, y_=0.0;
    // ...
  public:
    // ...
    Point2D(double x, double y): x_(x), y_(y) {}

    Point2D add(const Point2D & p){
       Point2D result(x_ + p.x_, y_ + p.y_);
        return result;
    }

};
```

In [ ]:
```
Point2D a(5.6, 4.3), b(4.5, 0.0);
Point2D c = a.add(b);
// wouldn't it be great to use Point2D c = a + b; ?
```

We will start with the assignment operator (`operator=`) Consider this example with int:

In [ ]:
```
int a, b, c = 4;
a = b = c; //a, b and c are now 4!
```

Why do we start with the assignment operator? It is very likely that you will need it: The assignment operator is the sibling of the copy constructor!

In [ ]:
```
// restart kernel to run
#include <iostream>

class Point2D {
  private:
    double x_=0.0, y_=0.0;
    // ...
  public:
    // ...
    Point2D(double x, double y): x_(x), y_(y) {}
    Point2D(const Point2D & rhs): x_(rhs.x_), y_(rhs.y_) {}

    Point2D& operator=(const Point2D & rhs){
      x_ = rhs.x_;
      y_ = rhs.y_;
      return *this;
    }

    void print(){
        std::cout << "x=" << x_ << ", y=" << y_ << std::endl;
    }
};
```

In [ ]:
```
Point2D p(4,5), q(2,0.8);
q.print();
q = p; // equivalent to q.operator=(p);
q.print();
```

Good programming practice: when you implement the operator=, also implement a copy constructor, and viceversa!

Question: can we use `Point2D& operator=(const Point2D & rhs)=default;` ?

✔ ✗

In [ ]:
```
Point2D a(3.4, 20);
Point2D b = a; //calls copy constructor
b = a; // calls operator=
```

Now let's implement the operator+ function for Point2D.

In [ ]:
```
#include <iostream>

class Point2D {
  private:
    double x_=0.0, y_=0.0;
    // ...
  public:
    // ...
    Point2D(double x, double y): x_(x), y_(y) {}
    Point2D(const Point2D & rhs): x_(rhs.x_), y_(rhs.y_) {}

    Point2D& operator=(const Point2D & rhs){
      x_ = rhs.x_;
      y_ = rhs.y_;
      return *this;
    }

    Point2D operator+(const Point2D & p){
       Point2D result(x_ + p.x_, y_ + p.y_);
       return result;
    }

    void print(){
        std::cout << "x=" << x_ << ", y=" << y_ << std::endl;
    }
};
```

In [ ]:
```
Point2D p1(3,4), p2(5.5,6);
Point2D p3(0,0);

p3 = p1 + p2;
p3.print();
```

Pre and post increment

The following code:

```
i++;
```

```
++i;
```

Sometimes results are the same, for example in:

```
for (int i =0; i < 10; i++) {
```

```
for (int i =0; i < 10; ++i) {
```

We will overload them so, you know exactly what happens, and why

```
int a = i++;
```

is not the same as

```
int a = ++i;
```

In [ ]:

```
class SpecialInt {
public:
  int spint;
  SpecialInt(int myint): spint(myint);
  SpecialInt(const SpecialInt & rhs)=default;
  SpecialInt & operator=(const SpecialInt & rhs)=default;

  SpecialInt& operator++() { // pre-increment ++i
    spint++;
    return *this;
  }

  SpecialInt operator++(int) { // post-increment i++
    SpecialInt temp = *this;
    spint++;
    return temp;
  }
}

SpecialInt a(3); // a = 3
SpecialInt b = ++a; // b = 4, a = 4, "b.SpecialInt(a.operator++())"
SpecialInt c = a++; // c = 4, a = 5, "b.SpecialInt(a.operator++(0))"
```

Question: in a for loop with SpecialInt (instead of int) which one would you use `++i` ?

✔ ✗

In [ ]: