- Solutions

Let's have a look at the solutions to the exercises we had in the last lesson.

WE'LL COVER THE FOLLOWING

Problem Statement 1: Solution

```
// vectorArithmeticExpressionTemplates.cpp

#include <cassert>
#include <iostream>
#include <vector>

template<typename T, typename Cont= std::vector<T> >
class MyVector{
  Cont cont;

public:
  // MyVector with initial size
  explicit MyVector(const std::size_t n) : cont(n){}

  // MyVector with initial size and value
  MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}

  // Constructor for underlying container
  explicit MyVector(const Cont& other) : cont(other){}

  // assignment operator for MyVector of different type
  template<typename T2, typename R2>
  MyVector& operator=(const MyVector<T2, R2>& other){
    assert(size() == other.size());
    for (std::size_t i = 0; i < cont.size(); ++i) cont[i] = other[i];
    return *this;
  }

  // size of underlying container
  std::size_t size() const{
    return cont.size();
```

```
  }

  // index operators

  T operator[](const std::size_t i) const{
    return cont[i];
  }

  T& operator[](const std::size_t i){
    return cont[i];
  }

  // returns the underlying data
  const Cont& data() const{
    return cont;
  }

  Cont& data(){
    return cont;
  }
};

// MyVector + MyVector
template<typename T, typename Op1 , typename Op2>
class MyVectorAdd{
  const Op1& op1;
  const Op2& op2;

public:
  MyVectorAdd(const Op1& a, const Op2& b): op1(a), op2(b){}

  T operator[](const std::size_t i) const{
    return op1[i] + op2[i];
  }

  std::size_t size() const{
    return op1.size();
  }
};

// MyVector - MyVector
template<typename T, typename Op1, typename Op2>
class MyVectorSub {
        const Op1& op1;
        const Op2& op2;

public:
        MyVectorSub(const Op1& a, const Op2& b) : op1(a), op2(b) {}

        T operator[](const std::size_t i) const {
                return op1[i] - op2[i];
        }

        std::size_t size() const {
                return op1.size();
        }
};

// elementwise MyVector * MyVector
template< typename T, typename Op1 , typename Op2 >
class MyVectorMul {
  const Op1& op1;
  const Op2& op2;
```

```
public:
  MyVectorMul(const Op1& a, const Op2& b ): op1(a), op2(b){}


  T operator[](const std::size_t i) const{
    return op1[i] * op2[i];
  }

  std::size_t size() const{
    return op1.size();
  }
};

// elementwise MyVector / MyVector
template< typename T, typename Op1, typename Op2 >
class MyVectorDiv {
        const Op1& op1;
        const Op2& op2;

public:
        MyVectorDiv(const Op1& a, const Op2& b) : op1(a), op2(b) {}

        T operator[](const std::size_t i) const {
                return op1[i] / op2[i];
        }

        std::size_t size() const {
                return op1.size();
        }
};

// function template for the + operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorAdd<T, R1, R2> >
operator+ (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
  return MyVector<T, MyVectorAdd<T, R1, R2> >(MyVectorAdd<T, R1, R2 >(a.data(), b.data()));
}

// function template for the - operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorSub<T, R1, R2> >
operator- (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
        return MyVector<T, MyVectorSub<T, R1, R2> >(MyVectorSub<T, R1, R2 >(a.data(), b.data(
}

// function template for the * operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorMul< T, R1, R2> >
operator* (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
  return MyVector<T, MyVectorMul<T, R1, R2> >(MyVectorMul<T, R1, R2 >(a.data(), b.data()));
}

// function template for the / operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorDiv< T, R1, R2> >
operator/ (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
        return MyVector<T, MyVectorDiv<T, R1, R2> >(MyVectorDiv<T, R1, R2 >(a.data(), b.data(
}

// function template for < operator
template<typename T>
std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
```

```
  std::cout << std::endl;
  for (int i=0; i<cont.size(); ++i) {
    os << cont[i] << ' ';
  }
  os << std::endl;
  return os;
}

int main(){

  MyVector<double> x(10,5.4);
  MyVector<double> y(10,10.3);

  MyVector<double> result(10);

  result = x + x + y * y - x + x + y / y;

  std::cout << result << std::endl;

}
```

Explanation

We have implemented the overloaded `-` operator (line 133 - 137), the overloaded `/` operator (line 147 - 151), and the overloaded `output` operator (line 154 - 162). Now, the objects `x`, `y`, and `result` feel like numbers.

Problem Statement 2: Solution

```
#include <cassert>
#include <functional>
#include <iostream>
#include <vector>

template<typename T, typename Cont= std::vector<T> >
class MyVector{
  Cont cont;

public:
  // MyVector with initial size
  explicit MyVector(const std::size_t n) : cont(n){}

  // MyVector with initial size and value
  MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}

  // Constructor for underlying container
  explicit MyVector(const Cont& other) : cont(other){}

  // assignment operator for MyVector of different type
  template<typename T2, typename R2>
  MyVector& operator=(const MyVector<T2, R2>& other){
    assert(size() == other.size());
```

```
    for (std::size_t i = 0; i < cont.size(); ++i) cont[i] = other[i];
    return *this;
  }


  // size of underlying container
  std::size_t size() const{
    return cont.size();
  }

  // index operators
  T operator[](const std::size_t i) const{
    return cont[i];
  }

  T& operator[](const std::size_t i){
    return cont[i];
  }

  // returns the underlying data
  const Cont& data() const{
    return cont;
  }

  Cont& data(){
    return cont;
  }
};

template<template<typename> class Oper, typename T, typename Op1 , typename Op2>
class MyVectorCalc{
  const Op1& op1;
  const Op2& op2;
  Oper<T> oper;

public:
  MyVectorCalc(const Op1& a, const Op2& b): op1(a), op2(b) {}

  T operator[](const std::size_t i) const{

    return oper(op1[i], op2[i]);
  }

  std::size_t size() const{
    return op1.size();
  }
};

// function template for the + operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorCalc<std::plus, T, R1, R2> >
operator+ (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
  return MyVector<T, MyVectorCalc<std::plus, T, R1, R2> >(MyVectorCalc<std::plus, T, R1, R2 >
}

// function template for the - operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorCalc<std::minus, T, R1, R2> >
operator- (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
        return MyVector<T, MyVectorCalc<std::minus, T, R1, R2> >(MyVectorCalc<std::minus, T,
}

// function template for the * operator
```

```
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorCalc<std::multiplies, T, R1, R2> >
operator* (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
  return MyVector<T, MyVectorCalc<std::multiplies, T, R1, R2> >(MyVectorCalc<std::multiplies,
}

// function template for the / operator
template<typename T, typename R1, typename R2>
MyVector<T, MyVectorCalc<std::divides, T, R1, R2> >
operator/ (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
        return MyVector<T, MyVectorCalc<std::divides, T, R1, R2> >(MyVectorCalc<std::divides,
}

// function template for << operator
template<typename T>
std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
  std::cout << std::endl;
  for (int i=0; i<cont.size(); ++i) {
    os << cont[i] << ' ';
  }
  os << std::endl;
  return os;
}

int main(){

  MyVector<double> x(10,5.4);
  MyVector<double> y(10,10.3);

  MyVector<double> result(10);

  result = x + x + y * y - x + x + y / y;

  std::cout << result << std::endl;

}
```

Explanation

In contrast to the previous example, in which each arithmetic operation such as addition (`MyVectorAdd`), subtraction (`MyVectorSub`), multiplication (`MyVectorMult`), or division (`MyVectorDiv`) is represented in a type, this improved version uses a generic binary operator (`MyVectorCalc`) in lines 52 – 69. This generic binary operator requires the concrete binary operator such as `std::plus` to become the concrete binary operator. The predefined function objects `std::plus`, `std::minus`, `std::multiplies`, and `std::divides` are part of the standard template library. You can think of them as a lambda-function representing the requested operation.

In the next lesson, we’ll study policy and traits in idioms and patterns.