- Exercises

Let's solve a few exercises in this lesson regarding expression templates.

WE'LL COVER THE FOLLOWING

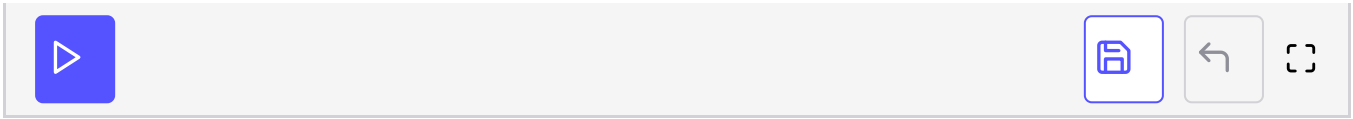
Problem Statement 1

Compare both programs containing `+` and `*` operators and study, in particular, the implementation based on expression templates.

- Extend the example to expression templates by adding support subtraction and the division operators for the `MyVector` class.

```
#include <iostream>
#include <vector>

template<typename T>
class MyVector{
  std::vector<T> 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){}

  // 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];
```

```
  }

};


// function template for the + operator
template<typename T>
MyVector<T> operator+ (const MyVector<T>& a, const MyVector<T>& b){
  MyVector<T> result(a.size());
  for (std::size_t s= 0; s <= a.size(); ++s){
    result[s]= a[s]+b[s];
  }
  return result;
}

// function template for the * operator
template<typename T>
MyVector<T> operator* (const MyVector<T>& a, const MyVector<T>& b){
  MyVector<T> result(a.size());
  for (std::size_t s= 0; s <= a.size(); ++s){
    result[s]= a[s]*b[s];
  }
  return result;
}

// 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;
}

// Implement subtraction and division operators here

int main(){
  // call these subtraction and division operators here
}
```



Problem Statement 2

The solution of the previous example `Vector Arithmetic Expression Templates` is too laborious. In particular, the classes `MyVectorAdd`, `MyVectorSub`, `MyVectorMul`, and `MyVectorDiv` have a lot in common. Try to simplify the program.

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

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


public:
  // MyVector with initial size
  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
  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();
  }
```

```
};

// 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();
  }
};

// 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, 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 < 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;

  std::cout << result << std::endl;
  */
}
```




In the next lesson, we'll look at the solutions to these exercises.