Expression Templates

In this lesson, we'll study expression templates in detail.

WE'LL COVER THE FOLLOWING

Expression Templates

Expression templates are “structures representing a computation at compile-time, which are evaluated only as needed to produce efficient code for the entire computation.”

Now we are at the center of lazy evaluation.

Lazy Evaluation

The story about lazy evaluation in C++ is quite short. That will change in C++20, with the ranges library from **Eric Niebler**. Lazy evaluation is the default in Haskell. Lazy evaluation means that an expression is only evaluated when needed.

This strategy has two benefits.

- Lazy evaluation helps you to save time and memory.
- You can define an algorithm on infinite data structures. Of course, you can only ask for a finite number of values at runtime.

Advantages:

- Creates a domain-specific language (DSL)
- Avoidance of temporary data structures

Disadvantages:

- Longer compile-times
- Advanced programming technique (template metaprogramming)

What problem do expression templates solve? Thanks to expression templates, we can get rid of superfluous temporary objects in expressions. What do we mean by superfluous temporary objects? To demonstrate that, let's look at the implementation of the class `MyVector` below.

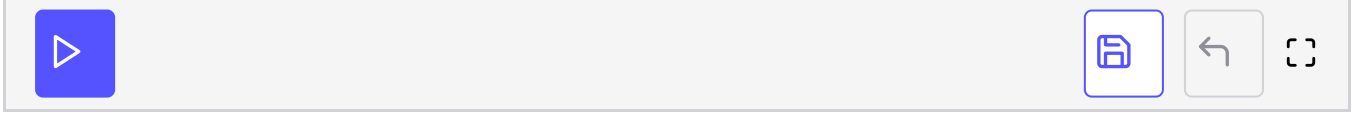
A First Naive Approach

`MyVector` is a simple wrapper for an `std::vector<T>`. The wrapper class has two constructors (line 12 and 15), we have a `size` function which returns its size (line 18 - 20), and the reading index operator (line 23 - 25) and writing index access (line 27 - 29).

Given below is the naive vector implementation:

```
// vectorArithmeticOperatorOverloading.cpp

#include <iostream>
#include <vector>

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

  // 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;
}

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;

}
```



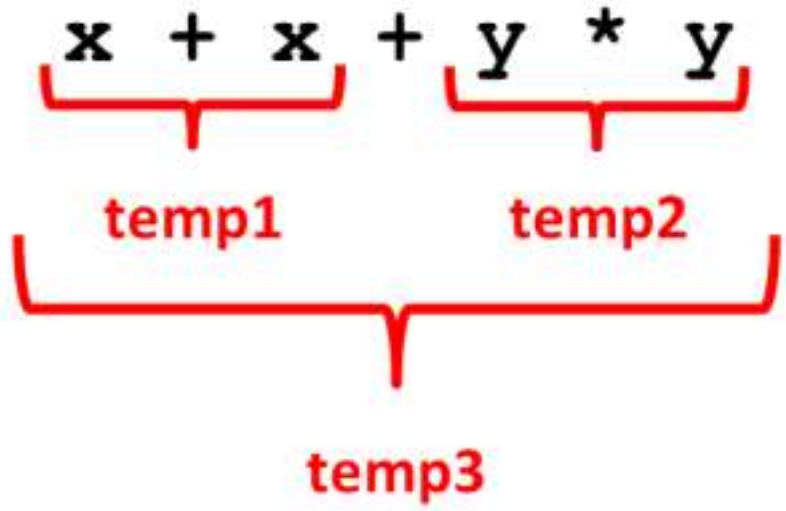
Thanks to the overloaded `+` operator (line 34 - 41), the overloaded `*` operator (line 44 - 51), and the overloaded `output` operator (line 54 - 62), the objects `x`, `y` and `result` feel like numbers.

Why is this implementation naive? The answer is in the expression `result = x+x + y*y`. In order to evaluate the expression, three temporary objects are needed to hold the result of each arithmetic subexpression.

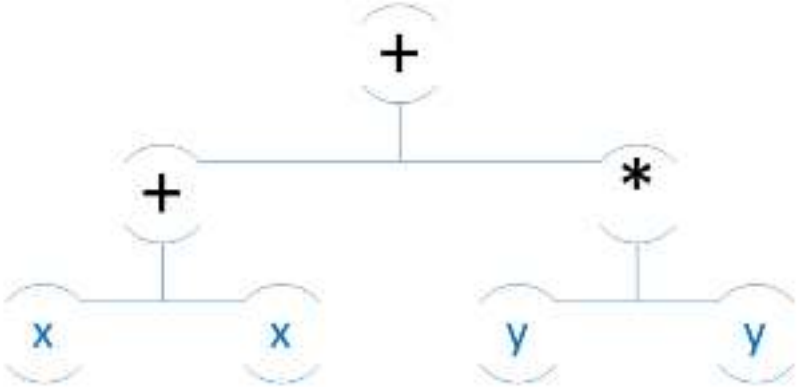
The Issue

```
result = x+x + y*y;
```

- Arithmetic expressions create many temporary objects



The Solution



The Idea

- The overloaded operators return proxy objects
- The final assignment `result[i] = x[i] + x[i] + y[i] * y[i]` triggers the

calculation

> No temporary objects are necessary.

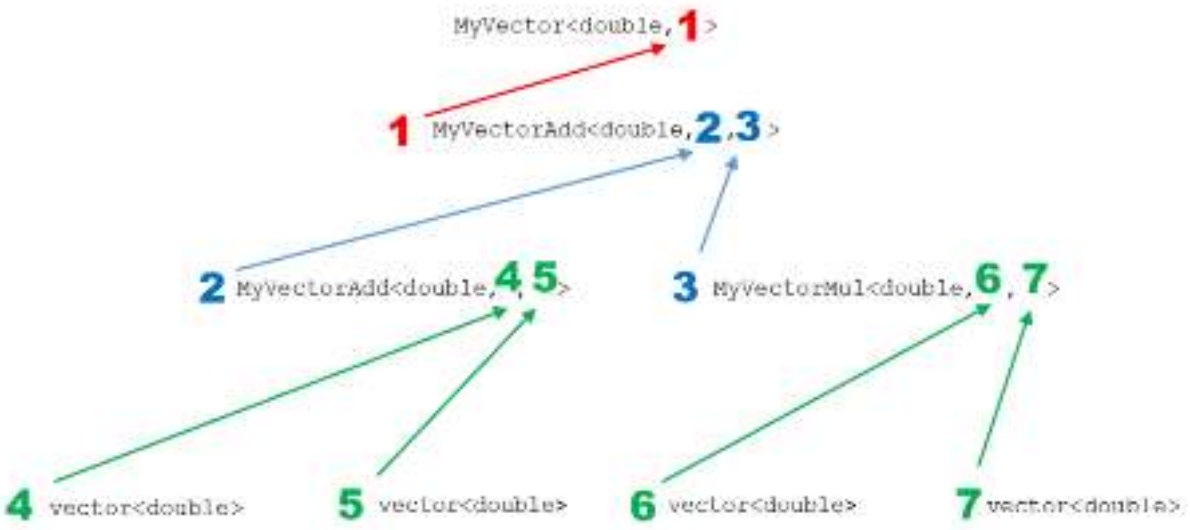
The Entire Magic

```
result[i] = x[i] + x[i] + y[i] * y[i];
```

Thanks to the compiler explorer you can see that no temporary objects are created. The expression is lazily evaluated in place.

```
        mov     rsi, rdx
        mov     rdi, rax
        call    MyVector<double, std::vector<double, std::allocator<double> > >& MyVector<double,
std::vector<double, std::allocator<double> > >::operator=<double, MyVectorAdd<double, MyVectorAdd<double,
std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> > >,
MyVectorMul<double, std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double>
> > > >(MyVector<double, MyVectorAdd<double, MyVectorAdd<double, std::vector<double, std::allocator<double>
>, std::vector<double, std::allocator<double> > >, MyVectorMul<double, std::vector<double,
std::allocator<double> >, std::vector<double, std::allocator<double> > > > const&)
        lea     rax, [rbp-160]
        mov     rdi, rax
```

A sharp view at the previous screenshot helps to unwrap the magic. Let's have a look at the sequence of calls:



In the next lesson, we'll look at some examples of expression templates.