1 Introduction

In this document we will look at a certain efficiency problem that occurs when creating classes that perform calculations that can be chained together (we will see what this means later on). This issue is very common, but in most cases it is left alone. Because of the relative complexity of solving this problem it isn't always worth it to solve it especially since it might affect how long the program takes to compile. However, sometimes the efficiency for these things matter a lot, and in that case you will have to bring out some more advanced tools to make it work. So keep that in mind.

2 Starting point

Consider the following (simple) implementation of a mathematical vector:

```
template <typename T, std::size_t dim = 3>
class Vector
{
public:
    template <typename... Ts>
    Vector(Ts &&... list)
        : data { std::forward<Ts>(list)... }
    { }

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

private:
    std::array<T, dim> data { };
};
```

This implementation focuses on accessing each element in the vector with an index. You could of course add other members, but for the purposes of this example this is enough. Now, suppose that this vector has the following operators defined:

```
template <typename T, std::size_t dim>
Vector<T, dim> operator+(Vector<T, dim> const& lhs,
                         Vector<T, dim> const& rhs);

template <typename T, std::size_t dim>
Vector<T, dim> operator*(Vector<T, dim> const& lhs,
                         T              const& rhs);
```

We can add two vectors of the same size (and type) and/or multiply a vector with a scalar. One might imagine more variants of these operators, as well as more operators. But for the purposes of this demonstration these are enough.

One thing to note here is that each of these operators will not modify any of the existing vectors, but will instead store the result in a new vector constructed from the operators (which is the expected behaviour).

Just for reference, here is a way to implement these operators:

```
template <typename T, std::size_t dim>
Vector<T, dim> operator+(Vector<T, dim> const& lhs,
                         Vector<T, dim> const& rhs)
{
    Vector<T, dim> result { };
    for (std::size_t i { 0 }; i < dim; ++i)
    {
        result[i] = lhs[i] + rhs[i];
    }
    return result;
}

template <typename T, std::size_t dim>
Vector<T, dim> operator*(Vector<T, dim> const& lhs,
                         T              const& rhs)
{
    Vector<T, dim> result { };
    for (std::size_t i { 0 }; i < dim; ++i)
    {
        result[i] = lhs[i] * rhs;
    }
    return result;
}
```

Now we have set the stage for the problem in question.

Disclaimer: Do note that writing code this way is not a problem itself, the simplicity of this implementation is a very good reason for writing this code. I usually recommend that you keep your code simple. But unfortunately the reality of the situation is that there is often a pay-off between simplicity and efficiency. So for the purposes of this demonstration we will prioritize efficiency over simplicity. Keep that in mind: you should avoid writing code as complex as the one presented below, unless it is absolutely necessary.

But before we get to a solution, let's discuss what the problem is.

3 The problem

The problem itself lies not as much with the class but rather how we use the class. Consider the following example:

```
using Vec = Vector<double, 3>;

Vec a { 1.2  , 3.4  , 5.6   };
Vec b { 7.8  , 9.10 , 11.12 };
Vec c { 13.14, 15.16, 17.18 };

Vec d { };
```

Now ask yourself the following questions: How many vectors are created here? What is the the least amounts of vectors that need to be created for this calculation to work?

In this question the answer to both questions is 4. `a`, `b`, `c` and `d`.

Now consider the same questions for this example (suppose we created `a`, `b`, `d` and `d` earlier, so we don't have to consider those):

```
d = a + b;
```

If you think about it you might consider that we copy the result of the addition `a + b` into `d`, but this should be easily solved by a good compiler (by using move-semantics). So the answer should be (if you have a good compiler) that we only create one new vector here.

So far everything seems totally fine. But what happens here?

```
d = a + b + c;
```

Let's start with `a + b + c`. Since we will assume that our compiler is smart enough to move the result from the expression into `d` it is enough to consider what happens in the expression.

Remember that addition is in reality evaluated like this:

```
((a + b) + c)
```

and `operator+` for our vectors will ALWAYS create a new vector. This means that we actually create 2 vectors: the resulting vector we get from `a + b` and then another one from when we add that with `c`.

This might not seem too bad (which, let's be honest: it's not), but what happens in cases like this?

```
d = (a + b) * 2.0 + (a + c) * 3.0 + (b + c) * 4.0;
```

This isn't a too far-fetched example since code like this do occur "in the wild".

So what is the answer? How many vectors are constructed? Well we have `a + b`, `a + c` and `b + c` so that is three right there. But that is not all. We also use multiplication on vectors which in turn construct a new vector. We multiply each of `a + b`, `a + c` and `b + c` with a scalar so that gives us a total of 6. But that is still not all, because we add it all together: and remember that each addition creates a new vector because of the way these operators are evaluated. The total number of vectors are 8. **In one expression!**

This might not seem too bad (which it really isn't for the case of three dimensional vectors), but imagine what would happen if these vector were much larger and therefore much more expensive to copy. We would have to sacrifice a lot of CPU cycles on performing copies as well as waste a lot of memory for storing these temporary vectors.

I'm sure you can think of an even more expensive calculation that can be performed with just `+` and `*`. But hopefully this example is large enough to get the point across.

There is a better way to calculate this expression: namely that we perform the following operation:

```
(a[i] + b[i]) * 2.0 + (a[i] + c[i]) * 3.0 + (b[i] + c[i]) * 4.0
```

for each index `i`. This will result in us creating no temporary vectors since we construct exactly one new vector from `a`, `b` and `c`: which is a huge gain compared to the way it is handled for the moment.

As it stands today, the compiler is not smart enough to realize this optimization, so we will have to take it into our own hands to try to implement this optimization ourselves.

The complete solution can be found in `vector.cc`. The rest of this document will walk you through how to construct this solution. But before we do that we have to take a step back and look at *lazy evaluation*.

4 Eager vs. Lazy evaluation

In a sense, we can ascribe the problem we have encountered to *eager evaluation*. That is, the fact that C++ evaluates every expression as soon as possible. The argument being that if C++ allowed expressions to be evaluated a bit later then the compiler might actually be able to do something about this issue.

The opposite of eager evaluation is called *lazy evaluation*. This is when an expression isn't evaluated until it is needed. The program is "lazy" and will only do the work once it is actually required.

To demonstrate the difference between eager and lazy evaluation consider this example:

```
double a { 2.5 };
double b { 3.5 };
double c { a + b };

std::cout << "My number: " << c << std::endl;
```

In C++ the value of `c` will be calculated immediately when it is initialized. This is because of eager evaluation. However, if C++ had lazy evaluation, it would instead evaluate the value of `c` in the absolute last second before it is needed. So in this case that would mean that a lazy evaluation would occur right when `c` is printed.

There are several languages that has lazy evaluation as its default, for example Haskell and F#. These languages has a completely different model, compared to C++, for how things are optimized as well as how things are handled during the execution of a program.

Eager evaluation is often cheaper to implement and lead to (in most cases) more efficient code. But in some cases (for example the vector that we presented earlier in this document) eager evaluation will lead to a significant slow down.

What is interesting with lazy evaluation is that we can look at the code without having any knowledge at all of the runtime context, and determine when an expression will be evaluated. Take this for example (under the assumption that we have lazy evaluation):

```
double sum { 0.0 };

double tmp;
while (std::cin >> tmp)
{
    sum += tmp;
}

std::cout << "The sum is: " << sum << std::endl;
```

Here we know that the actual value of the `sum` variable isn't needed until we try to print it. Therefore, without knowing how many iterations we will go through in the loop, or even what values are added to the sum, we will know exactly when the value will be calculated. This indicates that lazy evaluation actually takes place in the *compile-time*. I.e. that the compiler decides during compilation when an expression is evaluated.

C++ is eager in its evaluation. However, we are able to make decisions during the *compile-time* thanks to templates. Because of this, we are able to actually simulate lazy evaluation in C++.

5 Lazy evaluation in C++

Suppose we have the following class, which is meant to act as a simple integer:

```
class Integer
{
public:

    Integer(int value) : value{value} { }
    int get_value() const { return value; }

private:

    int value;
};
```

The goal for this section is to turn this into a lazily evaluated integer with addition and multiplication.

Notice that we can build quite complex expressions from just these two operators (`+` and `*`). For example:

```
(a + (b + c) * d + e) * f
```

where `a`, `b`, `c`, `d` and `f` are all objects of the type `Integer`.

In order to make this class lazily evaluated we will write code that "remembers" what expression it was generated from. We do this by being a bit more clever with the return type of the operators.

So instead of just returning an `Integer`, why not return a data type that communicates that this specific integer was generated from an addition or a multiplication?

Of course that will not handle expressions such as this one: `a + b * c` since it involves more than one operation. But, if we allow each expression to also store what expression the left- and right-hand-side was generated from, then we can encode the structure of the expression into the type.

Let's look at addition first. We introduce this class template:

```
template <typename LHS, typename RHS>
class Sum_Expression
{
public:

    Sum_Expression(LHS const& lhs, RHS const& rhs)
        : lhs{lhs}, rhs{rhs} { }

    int get_value() const
    {
        return lhs.get_value() + rhs.get_value();
    }

private:

    LHS lhs;
    RHS rhs;
};
```

Notice that we take two *arbitrary* types `LHS` and `RHS` and store an instance of each of these. These represent the expression that generated the respective operand.

Consider the following example:

```
a + b + c
```

where `a`, `b` and `c` are `Integer` objects. We know that it is evaluated as such:

```
((a + b) + c)
```

We want to represent this example as the following type:

```
Sum_Expression<Sum_Expression<Integer, Integer>, Integer>
```

The outer `Sum_Expression` has `LHS` as the type `Sum_Expression<Integer, Integer>` which represents the expression `a + b` and `RHS` as the type `Integer` which represents `c`. Since `LHS` and `RHS` are parameters to a `Sum_Expression` we know that it represents a usage of the addition operator.

So what does this have to do with lazy evaluation? Well notice that the actual value is not calculated until we call `get_value`, and once we do, it will recursively calculate all the subexpressions as well.

So, let's add the same thing but for multiplication:

```
template <typename LHS, typename RHS>
class Product_Expression
{
public:

    Product_Expression(LHS const& lhs, RHS const& rhs)
        : lhs{lhs}, rhs{rhs} { }

    int get_value() const
    {
        return lhs.get_value() * rhs.get_value();
    }

private:

    LHS lhs;
    RHS rhs;
};
```

Notice that it is very similar to `Sum_Expression`.

The only thing left to do is to create a connection between these classes and the corresponding operators. We do this by introducing the following operator overloads:

```
template <typename LHS, typename RHS>
Sum_Expression<LHS, RHS> operator+(LHS const& lhs,
                                   RHS const& rhs)
{
    return { lhs, rhs };
}

template <typename LHS, typename RHS>
Product_Expression<LHS, RHS> operator*(LHS const& lhs,
                                       RHS const& rhs)
{
    return { lhs, rhs };
}
```

So what we do here is that we take two arbitrary parameters[1] (`LHS` and `RHS`) and merge them together as either a `Sum_Expression` or `Product_Expression`. The resulting object will then evaluate the value of the expression once `get_value` is called on it.

Consider this example:

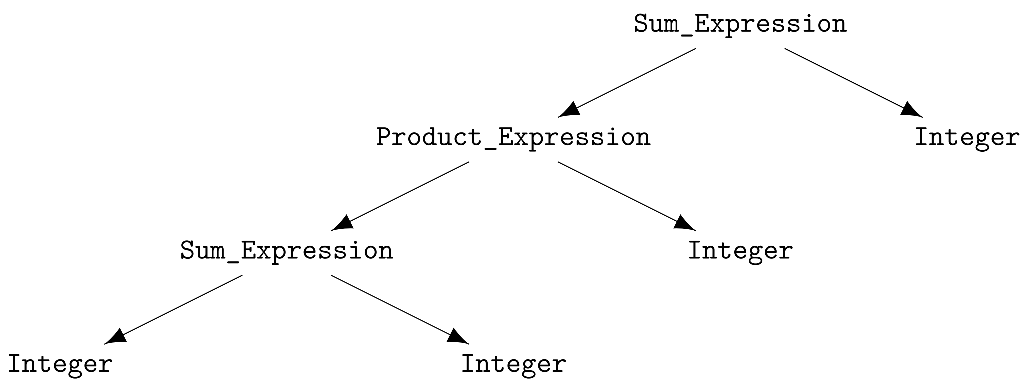
```
int main()
{
    Integer a { 5 };
    Integer b { 7 };
    Integer c { 8 };

    auto d { (a + b) * c + a };
}
```

What data type will `d` be? Well it will be:

```
Sum_Expression<Product_Expression<Sum_Expression<Integer,
                                                 Integer>,
                                  Integer>,
               Integer>
```

Which might be a bit hard to understand. But you can also represent this as a syntax tree, like this:



This means that we can evaluate `d` whenever we need it by calling `d.get_value()` and it will recursively calculate the value of each subexpression.

OK, so how do we use it? Well for now it can be a bit cumbersome:

```
Integer a { 1 };
Integer b { 2 };
Integer c { 3 };
auto d { a + b };

cout << d.get_value() << endl;
cout << (d + c).get_value() << endl;
```

[1]If you have a keen eye you might spot an issue here relating to ambigious function overloads. Suppose we have another class with similar operators, then we have two `operator+` that takes two arbitrary types and this will cause issues. See 7 for a solution.

Especially since we have to manually call `get_value` everytime we want the value. But we can make some simplifications that makes this work:

```
Integer a { 1 };
Integer b { 2 };
Integer c { 3 };
auto d { a + b };

cout << d << endl;
cout << d + c << endl;
```

By introducing the conversion operator `operator int` to all of our classes:

```
operator int() const
{
    return get_value();
}
```

This means that when we try to use our `Integer` it in a context where we are unable to perform lazy evaulation (for example when we try to print it) then the compiler can implicitly convert the `Integer` object to an `int`, and as a result it will evaluate the expression at that point.

The technique we used here, where we created a class template for each operator and stored subexpressions as template arguments, is called *expression templates*.

Note: If we use `auto` to declare variables we will run into issues, take for example:

```
auto d { a + b };
std::cout << d[0] << std::endl;
```

This example is wrong, since `d` is actually of the type `Sum_Expression<Vec, Vec>` instead of the desired type `Vec`. Therefore we should never use `auto` in this context and unfortunately there is no way for us forbid users from declaring it as `auto`. If you want to read more abuut this issue, look here: `http://eigen.tuxfamily.org/dox/TopicPitfalls.html` under the “C++11 and the auto keyword” header.

Now that we know how to simulate lazy evaluation with *expression templates*, let’s go back to our `Vector` and apply *expression templates*.

6 Expression Templates

Remember the original problem we had with our `Vector`, expressions such as:

```
(v + u) * 5 + w
```

creates way to many temporary vectors which has an impact on the efficiency of our code.

We can take inspiration from the lazy evaluation simulation and write our vector in such a way that it is guaranteed to not construct new vectors until it is absolutely necessary.

Just as we did previously, we add a class for each operator. Let's start with addition:

```
template <typename LHS, typename RHS>
class Vector_Sum
{
public:

    Vector_Sum(LHS const& lhs, RHS const& rhs)
        : lhs{lhs}, rhs{rhs} { }

    auto operator[](std::size_t i) const
    {
        return lhs[i] + rhs[i];
    }

private:

    LHS lhs;
    RHS rhs;

};

template <typename LHS, typename RHS>
Vector_Sum<LHS, RHS> operator+(LHS const& lhs, RHS const& rhs)
{
    return { lhs, rhs };
}
```

Here we just make another vector type which has the `operator[]`. But this time, we make the calculation of each element in the vector be lazily evaulated by only performing the addition inside the `operator[]`. Then we simply add an operator overload that takes two arbitrary types and produces a `Vector_Sum` from the passed in objects.

Notice that the return type of `operator[]` is `auto`, meaning the compiler will deduce the best type for the result.

Next let's do multiplication, which will be slightly different since only the left-hand-side will have an `operator[]`. So what we do is that we assume that the right-hand-side is something you can multiply with each element in the left-hand-side.

It will be something like this:

```
template <typename LHS, typename RHS>
class Vector_Product
{
public:

    Vector_Product(LHS const& lhs, RHS const& rhs)
        : lhs{lhs}, rhs{rhs} { }

    auto operator[](std::size_t i) const
    {
        return lhs[i] * rhs;
    }

private:

    LHS lhs;
    RHS rhs;

};

template <typename LHS, typename RHS>
Vector_Product<LHS, RHS> operator*(LHS const& lhs, RHS const& rhs)
{
    return { lhs, rhs };
}
```

So now we have changed our vector so that it only performs the calculations once we use `operator[]`. For example:

```
int main()
{
    using Vec = Vector<double, 3>;
    Vec a { 1.2  , 3.4  , 5.6   };
    Vec b { 7.8  , 9.10 , 11.12 };
    Vec c { 13.14, 15.16, 17.18 };
    auto d { (a + b) * 5.0 + c };

    for (int i {0}; i < 3; ++i)
    {
        // here we perform the calculations
        cout << d[i] << endl;
    }
}
```

So now the problem is fixed, right? *Unfortunately, no...*

If you have a keen eye you might have spotted that we now copy the vectors inside `Vector_Sum` and `Vector_Product`. Consider this example:

```
Vec a { 1, 2 };
Vec b { 3, 4 };
auto d { a + b };
```

The data type of `d` is:

```
Vector_Sum<Vec, Vec>
```

`operator+` will pass in `a` and `b` into the constructor of `Vector_Sum`, and what do we do with those? Let's take a look at the constructor again:

```
Vector_Sum(LHS const& lhs, RHS const& rhs)
    : lhs{lhs}, rhs{rhs} { }
```

So we initialize the data members `lhs` and `rhs` with the vectors. But those are stored inside the class as **copies**! So for each addition we actually copy each subexpression (which will contain vectors).

If you think about more complicated expression you will find that this is actually even *worse* than what we had before. So how do we fix this? Well fortunately it is qute an easy fix: we simply make `lhs` and `rhs` into `const` references:

```
template <typename LHS, typename RHS>
class Vector_Sum
{
public:

    Vector_Sum(LHS const& lhs, RHS const& rhs)
        : lhs{lhs}, rhs{rhs} { }

    // ...
private:

    LHS const& lhs;
    RHS const& rhs;

};
```

Now everything is nice and efficient. But, like everything, it comes with a price: now instead we have undefined behaviour. Why?

Well for our lazy evaluation to work we must make sure that all the data needed for it is available at the point where we evaluate the expression. However, when doing it like this, we will create a few temporary objects. Consider:

```
((a + b) + c)
```

Here we create a *temporary* `Vector_Sum` when we do `a + b` and another one when we add that with `c`. Since we bind `lhs` and `rhs` to `const` references we will have a problem once those temporary vectors are destructed (which is immediately after the expression has been constructed). So this means that all of the information is destroyed before we have the chance to use it. (Side note: even though we still create objects, notice that we never actually store any data besides references in any of those objects. This means that a decent compiler will be able to optimize these temporary objects away.)

So now that we have made `lhs` and `rhs` into references, we can no longer do lazy evaluation. But we can still evaluate the expression right before everything is destructed! Once the expression has been constructed we can immediately evaluate each element and store them in a `Vector` object. It would look something like this:

```
Vec a { 1, 2 };
Vec b { 3, 4 };
Vec c { 5, 6 };
Vec d { a + b + c };
```

Notice that we no longer use `auto` for `d`, instead we declare `d` as `Vec` (which, if you recall is an alias for `Vector<double, 3>`). Since `Vector` itself doesn't have any lazy evaluation we want to simply copy the result of the expression into `d`.

If we try to compile this we get an error because `Vector` doesn't have a constructor that takes `Vector_Sum`. This is also easily solved by adding the following constructor to `Vector`:

```
template <typename U>
Vector(U const& expression)
{
    for (std::size_t i {0}; i < dim; ++i)
    {
        data[i] = expression[i];
    }
}
```

Where we now take an *arbitrary type* as a parameter (but we assume that it has `operator[]`) and loop through each element and copy them into the `data` array.

However, this will clash with our existing constructor:

```
template <typename... Ts>
Vector(Ts &&... list)
    : data { std::forward<Ts>(list)... }
{ }
```

Which is needed for us to actually set the value of a vector.

So how can we solve this?

Well we replace our general constructor with an `std::initializer_list` constructor instead:

```
Vector(std::initializer_list<T> list)
{
    std::copy(list.begin(), list.end(), data.begin());
}
```

`std::initializer_list` is a special type that captures an arbitrary number of arguments (of the same type) into one list. This means that we now have made our constructor a bit better by only allowing it to take `T` which means that it will not clash with our newly added copy constructor.

Once we have made this change we have reached the implementation found in `vector.cc`. This implementation suffers from several issues which we will solve in the next section.

7 Cleaning up the interface

There are quite a few issues with the solution presented in `vector.cc`. Here is a list of a few things:

— The copy constructor for `Vector` matches any type of argument, but it only works for things relating to the vector. This means there might (as we saw) be some issues if we want other types of constructors as well.

— Our operators are way too general. They take arbitrary arguments which will clash with other similar operators. Suppose for example that we have both `Integer` and `Vector` in the same project: then which operator is used where? The compiler doesn't know.

There are some other issues which will not be solved in this document but that you are free to think about yourself:

— Adding two operators only makes sense if they have the same dimension, but our `operator+` doesn't consider that fact. This would require `Vector_Base` to keep track of the dimension somehow.

— If users declare their variables with `auto` there will be undefined behaviour in the program if the user initializes the variable from expressions involving our operators.

— There is no way for the user to determine if they want to activate this feature or not: what if they want eager evaluation?

— There is no way to store different types of vectors in the same container without introducing polymorphism. This is usually not a problem because we are mostly dealing with similar vectors. But if there is need for it, our vector wouldn't be able to do that.

The main issue that we will solve is that we have no singular way to represent a *vector* (be it a `Vector`, `Vector_Sum` or `Vector_Product`). We would want something similar to a base class which we can use to represent whichever vector type we want.

In order to solve these issues we will have to take a step back and make some changes to our vector. Let's begin with creating a base class that we call `Vector_Base`:

```cpp
template <typename Expression>
class Vector_Base
{
public:
    auto operator[](std::size_t i) const;
};
```

I will hold the implementation of `operator[]` for now, but you can think of it as similar to (but not the same as) a `virtual` function that we overload in the other implementations.

You might be wondering about the template parameter to this class. Well, think of it as representing the true type of this vector. Just as with polymorphic classes we have to store what the actual type of this vector is. You will see soon why this is useful.

Now, let's create `Vector` again, but with one major change:

```
template <typename T, std::size_t dim = 3>
class Vector : public Vector_Base<Vector<T, dim>>
{
public:

    Vector(std::initializer_list<T> list)
    {
        std::copy(list.begin(), list.end(), data.begin());
    }

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

private:

    std::array<T, dim> data { };
};
```

As you can see this is *almost* identical to the previous implementation but now we have a base class. It might look a bit strange to you since the base class is an instantiation of `Vector_Base` where we pass in `Vector` (which is the type we are defining!) as its template parameter. This might seem like it shouldn't work, but it does!

Since `Vector_Base` doesn't use its template parameter anywhere it will actually not cause any problems that we are passing in a derived class as its template parameter. This is because we never actually create an instance of `Expression` inside `Vector_Base`.

This concept, where we pass ourselves as the template parameter to our base class is surprisingly common. So common that it has a name: the *Curiously recurring template pattern* (*CRTP*).

But, the question remains: why?

In order to understand that, let's implement `Vector_Base::operator[]`:

```
template <typename Expression>
auto Vector_Base<Expression>::operator[](std::size_t i) const
{
  return static_cast<Expression const&>(*this)[i];
}
```

Let's explain this with an example. Suppose that we want to create a function that takes an arbitrary vector (which is quite a likely scenario). What type should that parameter be declared as?

We could declare it as `Vector<T, dim>`, but then we won't be able to utilize the expression templates technique to pass in an optimized version of expressions. The solution to this earlier was to just take an arbitrary template parameter and hope that it doesn't cause any ambiguities, which is not really good.

This is why we introduced `Vector_Base`, it allows us to only accept any of our vectors as a parameter (nothing more, nothing less). I.e. we can do this:

```
template <typename Expression>
auto first(Vector_Base<Expression> const& v)
{
    return v[0];
}
```

which is less general than our previous solution:

```
template <typename T>
auto first(T const& v)
{
    return v[0];
}
```

When we call `operator[]` in our new solution, it will call `Vector_Base::operator[]`. But we want it to call the "real" vectors version of `operator[]`. If we look at the definition of `Vector_Base::operator[]` we see the following is returned:

```
static_cast<Expression const&>(*this)[i]
```

Which means that we first convert ourselves into an `Expression`, which if you recall is the data type of the derived class (i.e. the actual type of our vector) and then we call `operator[]` on that.

This works because `Vector_Base` is the base class of `Expression`, meaning we can alway have a `Vector_Base` reference (or pointer) to `Expression`.

What we have managed to do here is create something very similar to a polymorphic base class with something akin to `virtual` functions. All because we store the information about what the actual type is as template parameter.

With this technique we can now solve those problems we presented at the beginning of this section. Let's introduce the following copy constructor to `Vector`:

```
template <typename Expression>
Vector(Vector_Base<Expression> const& other)
{
    for (std::size_t i{0}; i < dim; ++i)
    {
        data[i] = other[i];
    }
}
```

Now the constructor only accepts types that have `Vector_Base` as its base class (meaning, all of our vector types) and nothing more. This will make it a lot easier for us to introduce more constructors without causing any problems with ambiguity.

Our operators can also be cleaned up, for example:

```
template <typename LHS, typename RHS>
Vector_Sum<LHS, RHS> operator+(Vector_Base<LHS> const& lhs,
                               Vector_Base<RHS> const& rhs)
{
    return { lhs, rhs };
}
```

As long as we remember to make `Vector_Base<Vector_Sum<LHS, RHS>>` the base class of `Vector_Sum<LHS, RHS>` this operator will now only accept vector expressions.

You can see a complete implementation of this in `vector_crtp.cc` where we also implement `operator*` which only takes one vector (not two).

8 Closing words

This technique is frequently used by specifically Linear Algebra libraries, such as: Blaze, Boost uBLAS, Eigen and others.

It is also useful for big number libraries, parser libraries and loads of other libraries (your imagination is the limit!) where you construct objects from chaining operations.

This document served as an introduction to the concept of expression templates, but it is in no way a complete description. If you are interested in learning more you can read the Wikipedia article: `https://en.wikipedia.org/wiki/Expression_templates`

I can also highly recommend the book "C++ Templates - The Complete Guide, 2nd Edition" (`http://www.tmplbook.com/`) which, among a lot of other things, cover this exact topic. This book is sometimes called "The C++ Templates Bible" because it covers everything you might ever need to know about templates.