

EE380L Spring 2016, Project 3

valarray with Expression Templates

Implement the container `epl::valarray`. A valarray is essentially a vector that contains *values*, i.e., types that have the arithmetic operators defined. Your valarrays should have the following features:

1. (A) Arithmetic operations should be valid. That is, it is possible to add two valarrays, and the result should (at least conceptually) be another valarray.
 - a. The statement: $z = x \text{ op } y$ should set $z[k] = x[k] \text{ op } y[k]$ for all k and for any binary operator op . A reasonable subset of the binary and unary operators must be supported.
 - b. If the two operands are of differing length, the length of the result should be the minimum of the lengths of the two operands (ignore values past the end of the longer operand).
 - c. If one of the operands is a scalar, you should add (or whatever operation is implied) that scalar value to each of the elements in the valarray. In other words, you should implicitly expand the scalar to be a valarray of the appropriate length. Please do not actually create this implied valarray.
2. (A**/B) Lazy evaluation of expressions (using expression templates) are required. You should optimize the execution of statements like $z = (x * y) + w$; Where x, y, z, w are valarrays, so that only one loop gets executed and no temporary arrays are allocated. To clarify, when the compiler makes the call to `x.operator*(y)` x and y should not immediately be multiplied together. You should build up a representation of the expression, and only evaluate it when necessary (i.e., when the assignment operator, or the equivalent is called – watch out for that innocuous sounding “or equivalent” phrase, be sure you think through your use of expression templates). Expression templates must be transparent to the user – the semantics of your operator overloading should not reveal whether lazy evaluation or direct evaluation is being used.
 - a. It is acceptable to assume that the valarray components of an expression will not change before the expression is actually evaluated. For example, the following screwball case does not need to work correctly:
$$x = y + (z = z + 1);$$
3. (A*/B) You must provide a result of the appropriate type when valarrays with different element types are used. Promotion should be to the strictest type acceptable for the operation to occur – e.g., if a `valarray<int>` is added to a `complex<float>` the result should be a `valarray<complex<float> >`. If a `valarray<double>` is added to a `complex<float>` then the result should be a `valarray<complex<double> >`.

General notes about valarray:

- You must make `epl::vector<T>` a base class for `epl::valarray<T>`, and you must provide constructors for `void`, `std::initializer_list`, and `uint64_t`. Note that if you inherit the constructors from `epl::vector<T>`, then you can trivially meet this requirement. You are strongly encouraged to use inherited constructors

for this project. However, be mindful that Visual Studio 2013 does not support inherited constructors. **You are encouraged to develop and test your project with `std::vector<T>` in place of `epl::vector<T>` as your base class.** Simply switch to `epl::vector<T>` when you are ready to submit.

- You may **NOT** use the `valarray` from the STL in any way. You are prohibited from studying or reviewing the implement of `std::valarray`.
- You must define the following operations:
 1. (A) `push_back` -- inherited
 2. (A) `pop_back` -- inherited
 3. (A) `operator[]` -- inherited
 4. (A*/B) appropriate iterator and `const_iterator` classes (and `begin/end` functions) -- inherited
 5. (A) the binary operators `*,/,-,+`
 6. (A) a unary `-` (arithmetic negation)
 7. (B) all necessary functions to convert from one `valarray` type to another
 8. (A) a constructor that takes an initial size -- inherited
 9. (B*) a `sum()` function that adds all elements in the `valarray` using standard addition
 10. (B*) an `accumulate` function that adds all elements in the `valarray` using the given function object
 11. (B*) an *apply* member function that takes a unary function argument and returns (conceptually) a new `valarray` where the function has been applied to each element in the `valarray`. Of course, this *apply* method must follow all the rules for lazy evaluation (i.e., it won't return a real `valarray`, but rather some sort of expression template). HINT: function objects in the STL standard have nested types, including *result_type* which you might need to use.
 12. (B*) a *sqrt* member function that is implemented by passing a *sqrt* function object to the *apply* member function (just as I'm sure you implemented *sum* by passing *plus* to the *accumulate* method). The element type created from *sqrt* will either be **double** (for input `valarrays` that were **int**, **float** or **double** originally), or will be `std::complex<double>` for `valarrays` that were originally `std::complex<float>` or `std::complex<double>` originally.
- Everything in the list above (requirements for `valarray`) also applies to the result of a `valarray` expression. So, for example, `(x + y).sqrt()` should work. Perhaps more interestingly, a `const_iterator` must also be defined for the result of any expression, so `(x + y).begin()` should work. There are some subtleties to this as follows:
 - The return type of `(x + y).begin()` need not be `valarray::iterator`. We will only use "auto" to declare iterators, e.g., the following should work (B**):

```
auto p = (x + y).begin();
auto q = (x + y).end();
while (p != q) {
    cout << *p;
```

- ```

 ++p;
 }
 ○ The following code must work (with no allocations) (B*):
 valarray<int> x(10);
 valarray<double> y(10);
 for (auto const & p : x + y) {
 cout << p; // p should be a double
 }

```

### **Additional Requirements (due to the way we plan to test your valarray)**

1. (A) Please note that our testing will rely on the fact that there will be exactly one `mpl::vector<T>` for each `valarray<T>` that is created (and we'll be counting the number that are created).
2. (A) Be sure to implement operator<< for ostream and your valarray and also for your expression templates. I should be able to do "cout << x + y << endl" without allocating any memory (when x and y are valarrays). This is one of those "assignment operator or equivalent" things that I warned you about. In this case, you need to merely produce rvalues for each of the elements in the implied valarray. As such, you can preserve the illusion that x + y was computed and stored in a temporary, without actually allocating a temporary.

### **Hard Design Problems to Consider (but may not be worth many points, or may not be worth any points at all when we grade.)**

1. (B\*\*) Don't allow your templates to be instantiated unless the arguments are the correct type. E.g., your operator<< should get used if the argument is an Expression class, but shouldn't get used if I want to output some random Foo object. Your templates should not interfere with me if I want to write a template `<typename T1,typename T2> T1 operator+(T1 x, T2 y);`
2. (B\*\*) Solve this project using as few operator functions as possible. I'd like to say, "solve the project using the shortest solution you can come up with", but it's not really lines of code that matter, it's the number of functions and the number of template classes that you want to keep under control.

The use of `enable_if` can be very effective, but is not the only (and not necessarily the best) method for solving these "hard problems".