**Tuples**

The purpose of this assignment is to practice C++ class implementation for a generic programming, value semantic object with copy-on-write, and operator overloading. Create the type Tuple which implements the mathematical concept of a vector or n-tuple.

The following code represents how Tuple might be used:

double x[] = (1,2,3,4,5);

Tuple<int> i(5); // create a zero-filled n-tuple

Tuple<double> d(x,5) // create n-tuple from x[]

cout << i.size(); // number of elements

cout << d.magnitude(); // [Euclidean norm](http://en.wikipedia.org/wiki/Euclidean_vector#Length) of the vector

If **A** and **B** are Tuples of the same type, **x** is a value of the same type, and **i** is an integer, then the following operations should work:

A + B, A += B element-wise addition of zero-extended vectors,

resultant vector is size of longest operand

A \* B [dot product](http://en.wikipedia.org/wiki/Euclidean_vector#Dot_product) of two vectors,

result is scalar

x \* A, A \* x, A \*= x [scalar multiplication](http://en.wikipedia.org/wiki/Euclidean_vector#Scalar_multiplication) of vector and a scalar

A == B A exactly equals B, zero-extended as needed

A != B A does not equal B, zero-extended as needed

A[i] if 0≤i<size refers to ith element,

otherwise 0, both l- and r-value

Your solution should allow "low cost" value semantics as long as the underlying elements are not changed, but without requiring the value objects to be const. Using copy-on-write stategy and implementing a hand-body design to hide the reference counting. Any operation which changes (or potentially changes) an element should create a copy of the underlying elements.

In order to test you implementation, if **DEBUG** is defined, the implementation function should return a reference to an object of type TupleData, which should provide the following functions:

size() number of elements in the vector

useCount() number of "co-owners"

[i] if 0≤i<size refers to ith element, otherwise 0, both l- and r-value

You solution correctly manage access to private members and functions, be const correct, have appropriate order of operation time, implement operations only once, have canonical form (i.e., act like a value object), be commented, and so forth. The following code is a partial example of how your class will be tested.

Tuple<int> add(Tuple<int> a,Tuple<int> b,Tuple<int> c)

{

return a+b+c; // allocation for 2 anonymous objects, deallocation for 1 of them

}

int main()

{

Tuple<int> x(3); // allocation of 3 ints

Tuple<int> y(5); // allocation of 5 ints

x[0] = 2; // no allocation, only one "user" of data

x[1] = 3;

x[2] = 7;

y = x; // deallocation of 5 ints, share 3 ints

Tuple<int> z(x); // no allocation, another shared copy

z[3] = 9; // no allocation, z[3] is undefined so its a no-op

z[1] = 5; // copy/split occurs before write to element 1

z += x; // no allocation, z has its own copy

x = foo(x,y,x); // no memory allocation should occur due to copy construction

}