CSCI 241 Assignment 5
100 points

Assignment Overview

This program creates and implements a class to manipulate n-dimensional vectors. Don't try to picture these type of vectors in your brain. Just think of them as lists of numbers.

It is not necessary to have a working implementation of the previous assignment to complete this assignment. There are sufficient differences that starting over is necessary. However, if you understand how the Vector3 class works then implementing this assignment will be much easier.

A *driver program* is provided for this assignment to test your implementation. You don't have to write the tests.

Purpose

The purpose of this assignment is to give you some experience in dynamic memory allocation and all that that entails. It should also give you a bit of additional experience in operator overloading.

Program

You will need to write a single class for this assignment, the `VectorN` class. You will need to implement several methods and functions assoicated with this class.

The class should be implemented as two separate files. The class definition should be placed in an appropriately named header (`.h`) file. The implementations of the class methods and any other associated functions should be placed in a separate `.cpp` file for the class.

`class VectorN`

Data members

The `VectorN` class should contain the following `private` data members:

- a pointer to a `double`. I'll refer to this data member as the *vector array pointer*. It will be used to dynamically allocate an array of `double` (the *vector array*).
- an unsigned integer or `size_t` variable used to keep track of the number of elements in the vector array. I'll refer to this data member as the *vector capacity*.
- (The vector array's elements will always be completely filled with values, which means there is no need to track a separate *vector size*. The vector size will always be the same as the vector capacity.)

Methods and associated functions

The `VectorN` class should have the following methods (most of which are quite small):

- `VectorN::VectorN()`

  The default constructor for the `VectorN` class takes no arguments and should initialize a new `VectorN` object to an empty vector array with a capacity of 0. The required logic is:

  1. Set the vector capacity for the new object to 0.
  2. Set the vector array pointer for the new object to the special value `nullptr`.

- `VectorN::VectorN(const double values[], size_t n)`

  This constructor for the `VectorN` class should initialize a new `VectorN` object to the values stored in the array `values`. The required logic is:

  1. Set the vector capacity for the new object to `n`.
  2. If the vector capacity is 0, set the vector array pointer for the new object to `nullptr`. Otherwise, use the vector array pointer for the new object to allocate an array of `double`. The number of elements in the new string array should be equal to the vector capacity.
  3. Copy the elements of the array `values` into the vector array.

- `VectorN::VectorN(const VectorN& other)`

  This "copy constructor" for the `VectorN` class should initialize a new `VectorN` object to the same capacity and array contents as the existing `VectorN` object `other`. The required logic is:

  1. Set the vector capacity for the new object to the vector capacity of `other`.
  2. If the vector capacity is 0, set the vector array pointer for the new object to `nullptr`. Otherwise, use the vector array pointer for the new object to allocate an array of `double`. The number of elements in the new vector array should be equal to the vector capacity.
  3. Copy the contents of the vector array of `other` into the vector array of the new object. If `other` has a vector capacity of 0, this loop will exit immediately.

- `VectorN::~VectorN()`

  The destructor for the `VectorN` class can simply call the `clear()` method described below.

- `VectorN& VectorN::operator=(const VectorN& other)`

  This overloaded copy assignment operator should assign one `VectorN` object (the object `other`) to another (the object that called the method, which is pointed to by `this`). The required logic is:

  1. Check for self-assignment. If the address stored in the pointer `this` is the same as the address of the object `other`, then skip to the final step.
  2. Delete the vector array for the object pointed to by `this`.
  3. Set the vector capacity for the object pointed to by `this` to the vector capacity of `other`.
  4. If the vector capacity is 0, set the vector array pointer for the object pointed to by `this` to `nullptr`. Otherwise, use the vector array pointer to allocate an array of `double`. The number of elements in the new vector array should be equal to the vector capacity.
  5. Copy the contents of the vector array of `other` into the vector array of the object pointed to by `this`.
  6. Return `*this`.

- `void VectorN::clear()`

This method should properly set the instance back to a vector of zero elements. Delete the vector array, set the vector array pointer to `nullptr`, and set the vector capacity to 0.

- `size_t VectorN::size() const`

  Returns the size of the vector, which is equal to the vector capacity.

- `operator+`

  The addition operator should be overloaded to take two `VectorN`s and return a `VectorN`. The components of the result are computed by simply adding the components of the operands. For example (1, 2, 3) + (4, 5, 6) should have a result of (5, 7, 9). The operands should not be altered.

  If the two operands are of different capacities, then only the first `n` components of each vector should be used in the product where `n` is the capacity of the smaller of the two vectors. For example, (1, 2, 3) + (4, 5) should have a result of (5, 7).

  Implementation Hint: In any of the arithmetic operators, if you use a local `VectorN` variable to hold the result result, it may be necessary to first create an empty `VectorN` and then directly manipulate the vector array pointer and capacity to allocate memory for the vector array.

- `operator-`

  The binary subtraction operator should also be overloaded to take two `VectorN`s and return a `VectorN`. The result is the component-wise difference of the operands. For example (1, 2, 3) - (4, 6, 8) should have a result of (-3, -4, -5). As with addition, the result should have the capacity of the smaller operand.

- `operator*`

  The binary multiplication operator should be overloaded *three* times. The first form, called the *scalar product* takes two `VectorN`s and produces a single `double` value. The scalar product is computed by multiplying the corresponding components of the two vectors and adding the results. For example the scalar product of (1, 2, 3) and (4, 5, 6) is $(1 \cdot 4) + (2 \cdot 5) + (3 \cdot 6)$ which equals 4 + 10 + 18 which gives a final result of 32.

  If the two operands are of different capacities, then only the first `n` components of each vector should be used in the product where `n` is the capacity of the smaller of the two vectors.

  The other two overloaded multiplication operators allow multiplication of a `VectorN` with a `double` constant. For example, multiplying the vector (1, 2, 3) by 3 results in the vector (3, 6, 9). Two overloaded operators are needed here, one to handle multiplication of the vector by the constant in that order, the other to handle multiplication of the constant by the vector in that order. The results should be the same in both cases.

- `operator<<`

  The output operator should be overloaded so that a `VectorN` can be sent to the standard output. An empty vector (capacity 0) should be printed as `()`.

- `operator[]`

  The subscript operator should be overloaded to provide accessor methods for the class. The provided subscript indicates which value should be accessed from the dynamically-allocated array. For speed, no error checking needs to be done.

Don't forget that this operator needs to be overloaded twice, once for getting a value and once for setting a value.

- `double VectorN::at(int sub) const`

  This method is a variant of the read form of `operator[]` that provides some error checking.

  If `sub` is less than 0 or greater than or equal to the capacity of the vector array, this method should throw an `out_of_range` exception, like so:

  ```
  throw out_of_range("subscript out of range");
  ```

  Otherwise, the method should return element `sub` of the vector array.

  To make use of the existing standard exception class `out_of_range`, you will need to `#include <stdexcept>;` and code a `using` declaration for the class name. If you're confused about how to do this, take a look at the driver program.

- `double& VectorN::at(int sub)`

  This method is a variant of the write form of `operator[]` that provides some error checking.

  If `sub` is less than 0 or greater than or equal to the capacity of the vector array, this method should throw an `out_of_range` exception. Otherwise, it should return element `sub` of the vector array.

- `operator==`

  The equality operator should be overloaded to compare two `VectorN`s. The two vectors are considered equal only if they are componentwise equal. For example, (1, 2, 3) is equal to (1, 2, 3), but not to (4, 3, 2). All components must be equal. If the operands have different capacities, then the vectors are automatically not equal, regardless of the component values.

Output

A driver program, `assign5.cpp` is provided for this assignment. The purpose of a driver program is to test other pieces that you code. You do not need to write the driver program yourself. A copy of the driver program can also be found on `turing` at `/home/turing/t90kjm1/CS241/Code/Spring2015/Assign5/assign5.cpp`.

```
/*********************************************************************
   PROGRAM:    CSCI 241 Assignment 5
   PROGRAMMER: your name
   LOGON ID:   your z-ID
   DUE DATE:   due date of assignment

   FUNCTION:   This program tests the functionality of the VectorN
               class.
*********************************************************************/

#include <iostream>
```

```cpp
#include <stdexcept>
#include "VectorN.h"

using std::cout;
using std::endl;
using std::out_of_range;

int main()
   {
   int test = 1;

   cout << "\nTest " << test++ << ": Default constructor and printing\n" << endl;

   const VectorN v1;

   cout << "v1: " << v1 << endl;

   cout << "\nTest " << test++ << ": Array constructor and printing\n" << endl;

   double ar2[] = {1.0, 2.0, 3.0};

   const VectorN v2(ar2, 3);

   cout << "v2: " << v2 << endl;

   cout << "\nTest " << test++ << ": Clear and size\n" << endl;
   VectorN v3(ar2, 3);

   cout << "The size of v3: " << v3 << " is " << v3.size() << endl;

   v3.clear();

   cout << "After clearing, the size of v3: " << v3 << " is ";
   cout << v3.size() << endl;

   cout << "\nTest " << test++ << ": Subscripting\n" << endl;

   double ar3[] = {-1.0, -3.0, -5.0, 7.0};
   const VectorN v4(ar3, 4);

   cout << "v4: " << v4 << endl;
   cout << "v4[0]: " << v4[0] << "  v4[1]: " << v4[1] << endl;
   cout << "v4[2]: " << v4[2] << "  v4[3]: " << v4[3] << endl;

   VectorN v5(ar3, 4);
   v5[0] = 17;  v5[1] = 3;  v5[2] = 0;  v5[3] = -9;
   cout << "v5: " << v5 << endl;

   cout << "\nTest " << test++ << ": Copy constructor\n" << endl;

   const VectorN v6(ar2, 3);
   const VectorN v7 = v6;

   cout << "v6: " << v6 << " size: " << v6.size() << endl;
   cout << "v7: " << v7 << " size: " << v7.size() << endl;

   VectorN v8(ar3, 4);
   VectorN v9 = v8;

   cout << "v8: " << v8 << " size: " << v8.size() << endl;
   cout << "v9: " << v9 << " size: " << v9.size() << endl;
```

```
v8[1] = 300.0;
cout << "Changing..." << endl;
cout << "v8: " << v8 << " size: " << v8.size() << endl;
cout << "v9: " << v9 << " size: " << v9.size() << endl;

cout << "\nTest " << test++ << ": Assignment operator\n" << endl;
VectorN v10(ar3, 4);
VectorN v11;

cout << "v10: " << v10 << endl;
cout << "v11: " << v11 << endl;

v11 = v10;

v10[0] = 0.0;

cout << "v10: " << v10 << endl;
cout << "v11: " << v11 << endl;

cout << endl;

// Chained assignment
VectorN v12, v13;

cout << "v11: " << v11 << endl;
cout << "v12: " << v12 << endl;
cout << "v13: " << v13 << endl;

v13 = v12 = v11;

cout << "v11: " << v11 << endl;
cout << "v12: " << v12 << endl;
cout << "v13: " << v13 << endl;

cout << endl;

// Assignment to self

v13 = v13;

VectorN v14(ar2, 3);

cout << "v13: " << v13 << endl;

cout << "\nTest " << test++ << ": Addition and subtraction\n" << endl;

double ar4[] = {-2.0, 3.0, -1.0};
double ar5[] = {0, 1, 2, -3};

const VectorN v15(ar2, 3), v16(ar4, 3);
const VectorN v17(ar5, 4);

cout << "v13: " << v13 << endl;
cout << "v15: " << v15 << endl;
cout << "v16: " << v16 << endl;
cout << "v17: " << v17 << endl;

cout << endl;
cout << "v15 + v16 is " << v15 + v16 << endl;
cout << "v13 + v17 is " << v13 + v17 << endl;
```

```
cout << "v15 + v13 is " << v15 + v13 << endl;
cout << "v17 + v16 is " << v17 + v16 << endl;

cout << endl;
cout << "v15 - v16 is " << v15 - v16 << endl;
cout << "v13 - v17 is " << v13 - v17 << endl;
cout << "v15 - v13 is " << v15 - v13 << endl;
cout << "v17 - v16 is " << v17 - v16 << endl;

cout << "\nTest " << test++ << ": Vector multiplication\n" << endl;

cout << "The scalar product of " << v15 << " and " << v16 << " is ";
cout << v15 * v16 << endl;

cout << "The scalar product of " << v13 << " and " << v17 << " is ";
cout << v13 * v17 << endl;

cout << "The scalar product of " << v13 << " and " << v15 << " is ";
cout << v13 * v15 << endl;

cout << "The scalar product of " << v16 << " and " << v17 << " is ";
cout << v16 * v17 << endl;

cout << "\nTest " << test++ << ": Scalar multiplication\n" << endl;

double k = 2.345;

cout << v17 << " * " << k << " = " << v17 * k << endl;
cout << k << " * " << v17 << " = " << k * v17 << endl;

cout << "\nTest " << test++ << ": Equality\n" << endl;

double ar6[] = {1, 2, 2};
double ar7[] = {1, 2, -2};
double ar8[] = {1, 2, 2, 0};
double ar9[] = {1, 2, -2, 0};

const VectorN v18(ar6, 3), v19(ar7, 3);
const VectorN v20(ar8, 4), v21(ar9, 4);

cout << v18 << " and " << v18 << " are ";

if (v18 == v18)
   cout << "equal" << endl;
else
   cout << "not equal" << endl;

cout << v18 << " and " << v19 << " are ";

if (v18 == v19)
   cout << "equal" << endl;
else
   cout << "not equal" << endl;

cout << v18 << " and " << v20 << " are ";

if (v18 == v20)
   cout << "equal" << endl;
else
   cout << "not equal" << endl;
```

```
   cout << v21 << " and " << v19 << " are ";

   if (v21 == v19)
      cout << "equal" << endl;
   else
      cout << "not equal" << endl;

   cout << "\nTest " << test++ << ": Write form of at() method\n" << endl;

   VectorN v22(ar9, 4);

   try
      {
      v22.at(0) = 9.1;
      v22.at(1) = 3.97;
      v22.at(-1) = -7.43;
      }
   catch (out_of_range orex)
      {
      cout << "Caught "<< orex.what() << endl;
      }

   cout << "\nTest " << test++ << ": Read form of at() method\n" << endl;

   try
      {
      cout << "v22: (";
      for (unsigned i = 0; i <= v22.size(); i++)
         cout << v22.at(i) << ", ";
      cout << ")\n\n";
      }
   catch (out_of_range orex)
      {
      cout << endl << "Caught "<< orex.what() << endl;
      }

   return 0;
   }
```

Output from the correctly functioning driver program should look like the following:

```
Test 1: Default constructor and printing

v1: ()

Test 2: Array constructor and printing

v2: (1, 2, 3)

Test 3: Clear and size

The size of v3: (1, 2, 3) is 3
After clearing, the size of v3: () is 0

Test 4: Subscripting

v4: (-1, -3, -5, 7)
v4[0]: -1  v4[1]: -3
v4[2]: -5  v4[3]: 7
v5: (17, 3, 0, -9)
```

```
Test 5: Copy constructor

v6: (1, 2, 3) size: 3
v7: (1, 2, 3) size: 3
v8: (-1, -3, -5, 7) size: 4
v9: (-1, -3, -5, 7) size: 4
Changing...
v8: (-1, 300, -5, 7) size: 4
v9: (-1, -3, -5, 7) size: 4

Test 6: Assignment operator

v10: (-1, -3, -5, 7)
v11: ()
v10: (0, -3, -5, 7)
v11: (-1, -3, -5, 7)

v11: (-1, -3, -5, 7)
v12: ()
v13: ()
v11: (-1, -3, -5, 7)
v12: (-1, -3, -5, 7)
v13: (-1, -3, -5, 7)

v13: (-1, -3, -5, 7)

Test 7: Addition and subtraction

v13: (-1, -3, -5, 7)
v15: (1, 2, 3)
v16: (-2, 3, -1)
v17: (0, 1, 2, -3)

v15 + v16 is (-1, 5, 2)
v13 + v17 is (-1, -2, -3, 4)
v15 + v13 is (0, -1, -2)
v17 + v16 is (-2, 4, 1)

v15 - v16 is (3, -1, 4)
v13 - v17 is (-1, -4, -7, 10)
v15 - v13 is (2, 5, 8)
v17 - v16 is (2, -2, 3)

Test 8: Vector multiplication

The scalar product of (1, 2, 3) and (-2, 3, -1) is 1
The scalar product of (-1, -3, -5, 7) and (0, 1, 2, -3) is -34
The scalar product of (-1, -3, -5, 7) and (1, 2, 3) is -22
The scalar product of (-2, 3, -1) and (0, 1, 2, -3) is 1

Test 9: Scalar multiplication

(0, 1, 2, -3) * 2.345 = (0, 2.345, 4.69, -7.035)
2.345 * (0, 1, 2, -3) = (0, 2.345, 4.69, -7.035)

Test 10: Equality

(1, 2, 2) and (1, 2, 2) are equal
(1, 2, 2) and (1, 2, -2) are not equal
(1, 2, 2) and (1, 2, 2, 0) are not equal
```

```
(1, 2, -2, 0) and (1, 2, -2) are not equal

Test 11: Write form of at() method

Caught subscript out of range

Test 12: Read form of at() method

v22: (9.1, 3.97, -2, 0,
Caught subscript out of range
```

Implementation Hints

- Implement this similarly to the last assignment. Start off at the beginning of the driver program and try to get it working piece by piece. Maintain a working program as you go.

Other Points

- A `Makefile` is required. Same as always. Make sure it has appropriate rules for all the pieces involved.
- The class should have a header file for its class definition and a source code file for its implementation. Note that the driver program assumes that your header file is called `VectorN.h`. If you name your header file differently, you'll need to modify the driver program accordingly.
- All functions of the class that can be implemented as methods should be implemented as methods.
- Programs that do not compile on `turing/hopper` automatically receive 0 points.
- Submit your program using the electronic submission guidelines posted on the course web site.