Assignment 2 - Euclidean Vector

Change Log

- 27/06: List and vector type conversion examples updated in spec and client.cpp
- 28/06: Fixed client.sampleout path
- 28/06: Added exception conditions for at
- 28/06: Example usage of constructors updated from std::list to std::vector
- 28/06: Exception case added for GetEuclideanNorm member function
- 28/06: Typo fixed in 2.3, Addition changed to "Throw" from typo of "0"
- 01/07: Extra exception added to CreateUnitVector for case where euclieannormal is 0
- 03/07: Making clear that we will not test the case of multiplying two 0-dimension vectors together
- 04/07: Clarification in 2.1 that we will NOT be testing a constructor with no parameters
- 04/07: Test cases will use simple numbers such that any float point precision errors will not occur
- 05/07: Clarity that a moved-from object should have 0 dimensions
- 06/07: Added a section 3.9 to reiterate what we expect in terms of const-ness of your methods
- 06/07: Gave you a helping hand by further making clear the two (const and non-const) prototypes for .at
- 06/07: Added more clear information about both (1) explicit keyword, (2) 0-dimension vector math operations
- 06/07: Fixed a one word typo in 3.8 in the part about noexcept
- 06/07: Due to confusion on our end (sorry) we've actually given you pseudocode-near-implementation move constructor and move assignment within the spec. Given it's due in 7 days and people have been confused about this we don't want to create undue stress
- 06/07: "We will not be testing the case of multiplying two 0-dimension vectors together" moved from subtraction to multiplication
- 09/07: Function prototypes for majority of functions properly fleshed out. Please note: Not all information regarding const-ness is given. You will still have to make the appropriate functions const or non-const.
- 09/07: Move constructor example changed from

  ```
  aMove{a}
  ```

  to

  ```
  aMove{std::move(a)}
  ```

- 09/07: Parameters of iterator custructor have template typed added for full clarity
- 09/07: Typos in above changes fixed
- 10/07: Return type of operator= (copy and move) made more clear
- 10/07: Fixed "b.GetNumDimensions()" to be "c.GetNumDimensions()" for friend addition
- 10/07: Scalar division throw condition better explained
- 10/07: operator* now returns "double" instead of "const"
- 12/07: More info on default added
- 12/07: Section 4.4 added (automarking)
- 12/07: Section 3.10 added (catch2 test example)

1. Overview

Write a Euclidean Vector Class Library in C++, with its interface given in `euclidean_vector.h` and its implementation in `euclidean_vector.cpp`.

1.1 Aims

- Familiarity with C++ Classes
- Constructors
- Uniform initialisation
- Value semantics (Copy Control)
- Function Overloading
- Operator Overloading
- Friends
- Exception handling
- Separation of Interface from Implementation

2. Your task

2.1 Constructors

Name	Constructor	Description and Hints	Examples

Default Constructor	EuclideanVector(int)	A constructor that takes the number of dimensions (as a int) but no magnitudes, sets the magnitude in each dimension as 0.0. Hint: you may want to make this a delegating constructor to the next constructor below.
		This is the default constructor, with the default value being 1.
		You can assume the integer input will always be non-negative.
			`(1) EuclideanVector a{1};`
			`(2) int i {3};`
			`    EuclideanVector b{i};`
			Note: We will not be testing this c
			arguments)
			`EuclidenVector c;`

Constructor	EuclideanVector(int, double);	A constructor that takes the number of dimensions (as a int) and initialises the magnitude in each dimension as the second argument (a double). You can assume the integer input will always be non-negative.	`(1) EuclideanVector a{2, 4.0};`
			`(2) int x {3};`
			`    double y {3.24};`
			`    EuclideanVector b{x, y};`

Constructor	EuclideanVector(std::vector<double>::const_iterator, std::vector<double>::const_iterator)	A constructor (or constructors) that takes the start and end of an iterator to a std:vector and works out the required dimensions, and sets the magnitude in each dimension according to the iterated values.	`std::vector<double> v;`
			`EuclideanVector b{v.begin(),`

Copy Constructor	EuclideanVector(const EuclideanVector&)	`EuclideanVector aCopy{a};`	N/A
Move Constructor	EuclideanVector(EuclideanVector&&)	`EuclideanVector aMove{std::move(a)};`	N/A

Move constructor implementation

Due to confusion, this is provided to students free of charge! This assumes you use another private field that is a number "size". If your implementation varies, adjust accordingly.

```
EuclideanVector(EuclideanVector&& o) noexcept : magnitudes_{std::move(o.magnitudes_)}, size_{o.size_} {
  o.size_ = 0;
}
```

Example Usage

```
EuclideanVector a{1};      // a Euclidean Vector in 1 dimension, with default magnitude 0.0.
EuclideanVector b{2, 4.0};    // a Euclidean Vector in 2 dimensions with magnitude 4.0 in both dimensions

std::vector<double> l;
l.push_back(5.0);
l.push_back(6.5);
l.push_back(7.0);
EuclideanVector c{l.begin(), l.end()}; // a Euclidean Vector in 3 dimensions constructed from a vector of magnitud
```

Notes

- You may assume that all arguments supplied by the user are valid. No error checking on constructors is required.
- It's **very important** your constructors work. If we can't validly construct your objects, we can't test any of your other functions.

2.2. Destructor

You must explicitly declare the destructor as default.

For more info look here (https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/defaulted_functions_in_c_11?lang=en)

2.3. Operations

Name	**Operator**	Description	Examples	**Exception: Why thrown & what message**
Copy Assignment	EuclideanVector& operator=(const EuclideanVector&)	A copy assignment operator overload	`a = b;`	N/A

Move Assignment	EuclideanVector& operator=(EuclideanVector&&)	A move assignment operator	`a = std::move(b);`	N/A

Subscript	operator[]
	
	A const and non-const prototype is needed
		Allows to get and set the value in a given dimension of the Euclidean Vector. Hint: you may need two overloaded functions to achieve this requirement. **Note: It's a requirement you use asserts to ensure the index passed is valid.**	`double a {b[1]};`
			`b[2] = 3.0;`
				N/A

Addition	EuclideanVector& operator+=(const EuclideanVector&)	For adding vectors of the same dimension.	`a += b;`	**Given**: X = a.GetNumDimensions(), Y = b.GetNumDimensions() **When**: X != Y **Throw**: "Dimensions of LHS(X) and RHS(Y) do not match"
Subtraction	EuclideanVector& operator-=(const EuclideanVector&)	For subtracting vectors of the same dimension.	`a -= b;`	**Given**: X = a.GetNumDimensions(), Y = b.GetNumDimensions() **When**: X != Y **Throw**: "Dimensions of LHS(X) and RHS(Y) do not match"
Multiplication	EuclideanVector& operator*=(double)	For scalar multiplication, e.g. [1 2] * 3 = [3 6]	`a *= 3;`	N/A
Division	EuclideanVector& operator/=(double)	For scalar division, e.g. [3 6] / 2 = [1.5 3]	`a /= 4;`	**When**: b == 0 **Throw**: "Invalid vector division by 0"

Vector Type Conversion	explicit operator std::vector<double>()	Operators for type casting to a `std::vector`	`EucilideanVector a;`
			`auto vf = std::vector<double>{a};`
				N/A

List Type Conversion	explicit operator std::list<double>()	Operators for type casting to a `std::list`	`EucilideanVector a;`
			`auto lf = std::list<double>{a};`
				N/A

Move assignment implementation

Due to confusion, this is provided to students free of charge! This assumes you use another private field that is a number "size". If your implementation varies, adjust accordingly.

```
operator=(EuclideanVector&& o) noexcept {
  magnitudes_ = std::move(o.magnitudes_);
  size_ = o.size_;
  o.size = 0;
}
```

2.4. Methods

Prototype	Description	Usage	Exception: Why thrown & what message
double at(int) const	Returns the value of the magnitude in the dimension given as the function parameter	`a.at(1);`	**When**: For Input X: when X is < 0 or X is >= number of dimensions **Throw**: "Index X is not valid for this EuclideanVector object"

double& at(int)	Returns the reference of the magnitude in the dimension given as the function parameter	`a.at(1);`	**When**: For Input X: when X is < 0 or X is >= number of dimensions **Throw**: "Index X is not valid for this EuclideanVector object"
int GetNumDimensions()	Return the number of dimensions in a particular EuclideanVector	`a.GetNumDimensions();`	N/A
double GetEuclideanNorm()	Returns the Euclidean norm of the vector as a double. The Euclidean norm is the square root of the sum of the squares of the magnitudes in each dimension. E.g, for the vector [1 2 3] the Euclidean norm is sqrt(1*1 + 2*2 + 3*3) = 3.74.	`a.GetEuclideanNorm();`	**When**: this->GetNumDimensions() == 0 **Throw**: "EuclideanVector with no dimensions does not have a norm"
EuclideanVector CreateUnitVector()	Returns a Euclidean vector that is the unit vector of *this vector. The magnitude for each dimension in the unit vector is the original vector's magnitude divided by the Euclidean norm.	`a.CreateUnitVector();`	**When**: this->GetNumDimensions() == 0 **Throw**: "EuclideanVector with no dimensions does not have a unit vector" **When**: this->GetEuclideanNorm() == 0 **Throw**: "EuclideanVector with euclidean normal of 0 does not have a unit vector"

3.5. Friends

In addition to the operations indicated earlier, the following operations should be supported as friend functions. Note that these friend operations don't modify any of the given operands.

Name	**Operator**	**Description**	**Usage**	**Exception: Why thrown & what message**
Equal	bool operator==(const EuclideanVector&, const EuclideanVector&)	True if the two vectors are equal in the number of dimensions and the magnitude in each dimension is equal.	`a == b;`	N/A
Not Equal	bool operator!=(const EuclideanVector&, const EuclideanVector&)	True if the two vectors are not equal in the number of dimensions or the magnitude in each dimension is not equal.	`a != b;`	N/A
Addition	EuclideanVector operator+(const EuclideanVector&, const EuclideanVector&)	For adding vectors of the same dimension.	`a = b + c;`	**Given**: X = b.GetNumDimensions(), Y = c.GetNumDimensions() **When**: X != Y **Throw**: "Dimensions of LHS(X) and RHS(Y) do not match"
Subtraction	EuclideanVector operator-(const EuclideanVector&, const EuclideanVector&)	For substracting vectors of the same dimension.	`a = b - c;`	**Given**: X = b.GetNumDimensions(), Y = c.GetNumDimensions() **When**: X != Y **Throw**: "Dimensions of LHS(X) and RHS(Y) do not match"

Multiplication	double operator*(const EuclideanVector&, const EuclideanVector&)	For dot-product multiplication, returns a double. E.g., [1 2] * [3 4] = 1 * 3 + 2 * 4 = 11	`double c {a * b};`	**Given**: X = a.GetNumDimensions(), Y = b.GetNumDimensions() **When**: X != Y **Throw**: "Dimensions of LHS(X) and RHS(Y) do not match" **Note:** We will not be testing the case of multiplying two 0-dimension vectors together.
Multiply	EuclideanVector operator*(const EuclideanVector&, double)	For scalar multiplication, e.g. [1 2] * 3 = 3 * [1 2] = [3 6]. Hint: you'll obviously need two methods, as the scalar can be either side of the vector.	`(1) a = b * 3;` `(2) a = 3 * b;`	N/A
Divide	EuclideanVector operator/(const EuclideanVector&, double)	For scalar division, e.g. [3 6] / 2 = [1.5 3]	`double c;` `EuclideanVector a = b / c;`	**When**: c == 0 **Throw**: "Invalid vector division by 0"
Output Stream	std::ostream& operator<<(std::ostream&, const EuclideanVector&)	Prints out the magnitude in each dimension of the Euclidean Vector (surrounded by [and]), e.g. for a 3-dimensional vector: [1 2 3]	`std::cout << a;`	N/A

3.6. Internal Representation

Your Euclidean Vector is **required** to store the magnitudes of each dimension inside of a unique_ptr. This is a unique_ptr to a C-style double array. This has been added as part of the stub to your euclidean_vector.h file.

To create a dynamically allocated C-style double array and add it to a unique pointer, but not require any *direct* use of the new/malloc call, you can use the following:

```
std::unique_ptr<double[]> magnitudes_ = std::make_unique<double[]>(8); // 8 is an example
```

3.7. Throwing Exceptions

You are required to throw exceptions in certain cases. These are specified in the tables above. We have provided a EuclideanVectorError exception class for you to throw. You are welcome to throw other exceptions if you feel they are more appropriate.

Note: while the particular exception thrown does not matter, you are required to pass the strings specified in the tables above.

3.8. Other notes

You must:

- Include a header guard in euclidean_vector.h
- Use C++17 style and methods
- Make sure that *all appropriate member functions* are const qualified
- Leave a moved-from object in a state with 0 dimensions
- Must assume that addition, subtraction, multiplication, and division operations on two 0-dimension vectors are valid operations. In all cases the result should still be a 0-dimension vector.

You must not:

- Write to any files that aren't provided in the repo (e.g. storing your vector data in an auxilliary file)
- Add a main function euclidean_vector.cpp

You:

- Should try to mark methods that will not throw exceptions with
  ```
  noexcept
  ```
- Are not required to make any method explicit unless directly asked to in the spec.

3.9. Const Correctness

You must ensure that each operator (2.3) and method (2.4) appropriately either has:

- A const member function; or
- A non-const member function; or
- Both a const and non-const member function

Please think carefully about this. The function prototypes intentionally do not specify their constness, except for one exception, the **at()** operator. This has an explicit const and non-const prototype to help you out.

In most cases you will only need a single function, but in a couple of cases you will need both a const and non-const version.

3.10. Catch2 tests

Here is a sample and example of Catch2 tests to write

```
SCENARIO("Creation of unit vectors") {
  WHEN("You have two identical vectors") {
    EuclideanVector a{2};
    a[0] = 1;
    a[1] = 2;
    EuclideanVector b{2};
    b[0] = 1;
    b[1] = 2;
    THEN("Get their unit vectors") {
      EuclideanVector c{a.CreateUnitVector()};
      EuclideanVector d{b.CreateUnitVector()};
      REQUIRE(c == d);
    }
  }
}

SCENARIO("Accessing dimension that doesn't exist") {
  WHEN("You try to split an empty string") {
    EuclideanVector a{2};
    a[0] = 1;
    a[1] = 2;
    THEN("You get a single empty result") {
      REQUIRE_THROWS_WITH(a.at(2), "Index 2 is not valid for this EuclideanVector object");
    }
  }
}
```

4. Getting Started

If you haven't done so already, clone the repository:

```
$ git clone https://github.com/cs6771/comp6771 comp6771
```

Then navigate to the **assignments/ev** directory

All of the files you need are in this directory. Here is a list of files that and a description of their purpose:

File	Description
client.(cpp|sampleout)	A simple use case of a client using your euclidean_vector **Note: Do NOT modify this file. We will potentially update it. If you want to modify it, make a local copy first.**
euclidean_vector.(cpp|h)	Your euclidean vector interface and implementation. The reference solution is around the 500 lines of code mark (to give you a sense of the size)
euclidean_vector_test.cpp	Your tests for your euclidean vector file
BUILD	Build file containing build, test, dependency instructions

4.2. Reference Solution & Time Limits

Due to the simple nature of this data type, there will be performance based measurements

Each test will still have a time limit to run (1 second max), but unless you do something completely insane this will be more than enough time for all of your operations to complete.

4.3. Running your assignment

4.3.1. Running a basic use case

You can run your code against a basic (non-testing) use case to get a better sense of the behaviour

From your project directory:

```
$ bazel build //assignments/ev:client
```

```
$ ./bazel-bin/assignments/ev/client | diff ./assignments/ev/client.sampleout -
```

4.3.2. Running your tests

```
$ bazel build //assignments/ev:euclidean_vector_test
```

```
$ bazel run //assignments/ev:euclidean_vector_test
```

4.4. Autotests

On a CSE machine you can run

```
6771 evtest
```

to test your code automatically. Note: euclidean_vector.cpp and euclidean_vector.h must be in that directory.

4.5. Assessment

This assignment will contribute 15% to your final mark.

The assessment for the assignment recognizes the difficulty of the task, the importance of style, and the importance of appropriate use of programming methods (e.g. using while loops instead of a dozen if statements).

60%	**Correctness** The correctness of your program will be determined automatically by tests that we will run against your program. We will create a series of test files that contain their own main function and are compiled with your Euclidean vector file(s).
20%	**Your tests** You are required to write your own tests to ensure your program works. You will write tests in `euclidean_vector_test.cpp` . At the top of this file you will also include a block comment to explain the rational and approach you took to writing tests. Please read the Catch2 tutorial (https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md) or review lecture/tutorial content to see how to write tests. Tests will be marked on several factors. These include, but are not limited to • Correctness - an incorrect test is worse than useless. • Coverage - your tests might be great, but if they don't cover the part that ends up failing, they weren't much good to you. • Brittleness - If you change your implementation, will the tests need to be changed (this generally means avoiding calling functions specific to your implementation where possible - ones that would be private if you were doing OOP). • Clarity - If your test case failed, it should be immediately obvious what went wrong (this means splitting it up into appropriately sized sub-tests, amongst other things).
15%	**C++ style** Your adherence to good C++ style. This is **not** saying that if you conform to the style guide you will receive full marks for this section. This 20% is also based on how well you use modern C++ methodologies taught in this course as opposed to using backwards-compatible C methods. Examples include: Not using primitive arrays and not using pointers.

5%	**cpplint and clang-format**
	In your project folder, run the following commands on all C++ files you submit:
	`$ clang-format -i /path/to/file.cpp && python cpplint.py /path/to/file.cpp`
	If the program outputs the following, you will receive full marks for this section (5%). Otherwise you will receive no marks.
	`Done processing assignments/ev/euclidean_vector.cpp`
	`Total errors found: 0`

The following actions will result in a 0/100 mark for EuclideanVector, and in some cases a 0 for COMP6771:

- Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
- Submitting any other person's work. This includes joint work.
- Submitting another person's work without their consent.

The lecturer may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

4.6. Originality of Work

The work you submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such submissions.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct.

Do not provide or show your assignment work to any other person — apart from the teaching staff of COMP6771. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note you will not be penalized if your work has the potential to be taken without your consent or knowledge.

4.7. Submission

This assignment is due **Saturday 13th of July, 22:59:59**. Submit the assignment using this *give* command:

```
give cs6771 euclideanvector euclidean_vector.cpp euclidean_vector.h euclidean_vector_test.cpp
```

4.8. Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 1%. For example if an assignment worth 76% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 95%). If the same assignment was submitted 30 hours late it would be awarded 70%, the maximum mark it can achieve at that time.