Assignment 2 - Euclidean Vector

Please `git pull` frequently to get the latest changes.

Change Log

- 17/06: Initial release

The Task

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

We have outlined all key parts of this class below that should be implemented.

1. Constructors

You may have to scroll horizontally to view these tables

Name	Constructor	Description and Hints	Examples
Default Constructor	`euclidean_vector()`	A constructor that generates a euclidean vector with a dimension of 1 and magnitude of 0.0. You can assume the integer input will always be non-negative.	`(1) auto a = comp6771::euclidean_vector`

Single-argument Constructor	`explicit euclidean_vector(int)`	A constructor that takes the number of dimensions (as an int) but no magnitudes, sets the magnitude in each dimension as 0.0. You can assume the integer input will always be non-negative.	`(1) auto a = comp6771::euclidean_vector`
			`(2) int i {3};`
			`auto b = comp6771::euclidean_vector`

Constructor	`euclidean_vector(int, double);`	A constructor that takes the number of dimensions (as an `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) auto a = comp6771::euclidean_vector`
			`4.0);`
			`(2) auto x = int{3};`
			`auto y = double{3.24};`
			`auto b = comp6771::euclidean_vector`

Constructor	`euclidean_vector(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<double>` and works out the required dimensions, and sets the magnitude in each dimension according to the iterated values.	`std::vector<double> v;`
			`auto b =`
			`comp6771::euclidean_vector(v.begin(),v.`

Constructor	`euclidean_vector(std::initializer_list<double>)`	A constructor that takes an initialiser list of `double`s to populate vector magnitudes. You will have to do your own research to implement this one.	`auto b = comp6771::euclidean_vector{1.0` `3.0};`
Copy Constructor	`euclidean_vector(euclidean_vector const&)`		`auto a = comp6771::euclidean_vector(a);`
Move Constructor	`euclidean_vector(euclidean_vector&&)`		`auto aMove =` `comp6771::euclidean_vector(std::move(a)`

Example Usage

```
auto a = comp6771::euclidean_vector(1);      // a Euclidean Vector in 1 dimension, with default magnitude 0.0.
auto b = comp6771::euclidean_vector(2, 4.0); // a Euclidean Vector in 2 dimensions with magnitude 4.0 in both
dimensions

auto v = std::vector<double>{5.0, 6.5, 7.0};
auto c = comp6771::euclidean_vector(v.begin(), v.end()); // a Euclidean Vector in 3 dimensions constructed from a
vector of magnitudes
```

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. Destructor

You must explicitly declare the destructor as default.

For more info look here.

3. Operations

Name	Operator	Description	Examples	Exception: Why thrown & what message
Copy Assignment	`euclidean_vector& operator=(euclidean_vector const&)`	A copy assignment operator overload	`a = b;`	N/A
Move Assignment	`euclidean_vector& operator=(euclidean_vector&&)`	A move assignment operator	`a = std::move(b);`	N/A
Subscript	`operator[]` A const and non-const declaration is needed	Allows to get and set the value in a given dimension of the Euclidean vector. Hint: you may need two overloadeds to achieve this requirement.	`double a {b[1]};` `b[2] = 3.0;`	N/A
Unary plus	`euclidean_vector operator+()`	Returns a copy of the current object.	`+a`	N/A

Negation	`euclidean_vector operator-()`	Returns a copy of the current object, where each scalar value has its sign negated.	`auto const actual = comp6771::euclidean_vector{-6, 1};`
			`auto const expected = comp6771::euclidean_vector{6, -1};`
			`CHECK(expected == -actual);`
				N/A

Compound Addition	`euclidean_vector& operator+=(euclidean_vector const&)`	For adding vectors of the same dimension.	`a += b;`	**Given**: `X = a.dimensions(), Y = b.dimensions()` **When**: `X != Y` **Throw**: "Dimensions of LHS(`X`) and RHS(`Y`) do not match"
	`euclidean_vector&`			**Given**: `X = a.dimensions(), Y = b.dimensions()`

Compound Subtraction	`operator-=(euclidean_vector const&)`	For subtracting vectors of the same dimension.	`a -= b;`	**When**: `X != Y` **Throw**: "Dimensions of LHS(`X`) and RHS(`Y`) do not match"
Compound Multiplication	`euclidean_vector& operator*=(double)`	For scalar multiplication, e.g. `[1 2] * 3 = [3 6]`	`a *= 3;`	N/A
Compound Division	`euclidean_vector& 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>()` Const.	Operators for type casting to a `std::vector`	`auto const a = comp6771::euclidean_vector{0.0, 1.0, 2.0}; auto const vf = static_cast<std::vector<double>>(a);`	N/A
List Type Conversion	`explicit operator std::list<double>()` Const.	Operators for type casting to a `std::list`	`auto const a = comp6771::euclidean_vector{0.0, 1.0, 2.0}; auto lf = static_cast<std::list<double>>(a);`	N/A

4. Member Functions

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 euclidean_vector 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 euclidean_vector object"
`int dimensions()`	Return the number of dimensions in a particular euclidean_vector	`a.dimensions();`	N/A

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==(euclidean_vector const&, euclidean_vector const&)`	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!=(euclidean_vector const&, euclidean_vector const&)`	True if the two vectors are not equal in the number of dimensions or the magnitude in any dimension is not equal.	`a != b;`	N/A
Addition	`euclidean_vector operator+(euclidean_vector const&, euclidean_vector const&)`	For adding vectors of the same dimension.	`a = b + c;`	**Given**: `X = b.dimensions()`, `Y = c.dimensions()` **When**: `X != Y` **Throw**: "Dimensions of LHS(`X`) and RHS(`Y`) do not match"
Subtraction	`euclidean_vector operator-(euclidean_vector const&, euclidean_vector const&)`	For substracting vectors of the same dimension.	`a = b - c;`	**Given**: `X = b.dimensions()`, `Y = c.dimensions()` **When**: `X != Y` **Throw**: "Dimensions of LHS(`X`) and RHS(`Y`) do not match"
Multiply	`euclidean_vector operator*(euclidean_vector const&, double)`	For scalar multiplication, e.g. `[1 2] * 3 = 3 * [1 2] = [3 6]`. Hint: you'll need two operators, as the scalar can be either side of the vector.	`(1) a = b * 3;` `(2) a = 3 * b;`	N/A
Divide	`euclidean_vector operator/(euclidean_vector const&, double)`	For scalar division, e.g. [3 6] / 2 = [1.5 3]	`auto b = comp6771::euclidean_vector(3, 3.0);` `auto c = double{2.0};` `auto a = b / c;`	**When**: `c == 0` **Throw**: "Invalid vector division by 0"
Output Stream	`std::ostream& operator<<(std::ostream&, euclidean_vector const&)`	Prints out the magnitude in each dimension of the Euclidean vector (surrounded by `[` and `]`), e.g. for a 3-dimensional vector: `[1 2 3]`. **Note:** When printing the magnitude, simple use the double << operator.	`std::cout << a;`	N/A

6. Utility functions

The following are functions that operate on a Euclidean vector, but shouldn't be a part of its interface. They *may* be friends, if you need access to the implementation, but you should avoid friendship if you can.

Name	Description	Usage	Exception: Why thrown & what message
`auto euclidean_norm(euclidean_vector const& v) -> double;`	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`. If `v.dimensions() == 0`, the result is 0.	`comp6771::euclidean_norm(a);`	N/A
`auto unit(euclidean_vector const& v) -> euclidean_vector;`	Returns a Euclidean vector that is the unit vector of `v`. The magnitude for each dimension in the unit vector is the original vector's magnitude divided by the Euclidean norm.	`comp6771::unit(a);`	**When**: `v.dimensions() == 0` **Throw**: "euclidean_vector with no dimensions does not have a unit vector" **When**: `comp6771::euclidean_norm(v) == 0` **Throw**: "euclidean_vector with zero euclidean normal does not have a unit vector"
`auto dot(euclidean_vector const& x, euclidean_vector const& y) -> double`	Computes the dot product of `x` · `y`; returns a `double`. E.g., `[1 2] · [3 4] = 1 * 3 + 2 * 4 = 11`	`auto c = double{comp6771::dot(a, b)};`	**Given**: `X = a.dimensions()`, `Y = b.dimensions()` **When**: `X != Y` **Throw**: "Dimensions of LHS(`X`) and RHS(`Y`) do not match"

The Euclidean norm should only be calculated when required and ideally should be cached if required again. We may run test cases on large vectors calculating the Euclidean norm many times. Hint: consider using a mutable data member where appropriate in conjunction with another data member to appropriately cache the euclidean norm. This is done for performance reasons.

7. Compulsory Data Members

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.

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`/`std::malloc` call, you can use the following:

```
// ass2 spec requires we use double[]
// NOLINTNEXTLINE(modernize-avoid-c-arrays)
auto magnitudes_ = std::make_unique<double[]>(8); // 8 is an example
```

Please note, the theory for `unique_ptr` will be covered in week 5. Until that point, there will be *parts* of the assignment (e.g. move constructors, copy constructors) that you may struggle to implement. However, before week 5 lectures you are able to implement many other functions. That is because the `unique_ptr` is in many ways an alias for a raw pointer - i.e. you can treat `magnitudes_` like a raw pointer

For example:

```
this->magnitudes_[0] += other.magnitudes_[0]
```

8. Throwing Exceptions

You are required to throw exceptions in certain cases. These are specified in the tables above. We have provided a `euclidean_vector` 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. In these strings, please use common sense to substitute values like X and Y for their actual numerical values

9. Other notes

You must:

- Include a header guard in `euclidean_vector.h`
- Use C++20 style and methods where appropriate
- Make sure that *all appropriate member functions* are `const`-qualified
- Leave a moved-from object in a state with `0` dimensions
- Implement this class within the **comp6771** namespace
- Unless otherwise specified, 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.
- We're asking you to implement `operator!=` because you'll see it in a lot of production codebases, and it's important that you know how to write it correctly
- Where possible, avoid using C-style for loops when an appropriate STL algorithm can be used on an iterator that wraps a C-style array

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`
- Use `new` and `delete` commands explicitly to allocate or deallocate memory (we will discuss how to avoid this in the smart pointers lecture)
- Use any STL containers as part of your implementation
- Do any unnecessary pointer arithmetic, and use appropriate abstractions when you can. The best and simplest way to explain this is that if you're using `*` anywhere in your code besides for `*this` then you're *probably* doing something wrong.

You:

- must mark member functions that will never throw exceptions with `noexcept`
- Are not required to make any member function explicit unless directly asked to in the spec.

10. `const`-correctness

You must ensure that each operator (3.) and method (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 declarations intentionally do not specify their constness, except for one exception, the `at()` operator. This has an explicit `const` and non-`const` declaration 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.

Testing

Here is a sample and example of Catch2 tests to write:

```
TEST_CASE("Creation of unit vectors") {
  SECTION("You have two identical vectors") {
    auto a = comp6771::euclidean_vector(2);
    a[0] = 1;
    a[1] = 2;
    auto b = comp6771::euclidean_vector(2);
    b[0] = 1;
    b[1] = 2;

    auto c = comp6771::unit(a);
    auto d = comp6771::unit(b);
    REQUIRE(c == d);
  }
}
```

To test exceptions, you may find functions like
`REQUIRE_THROWS_WITH`
useful.

Getting Started

If you haven't done so already, clone this repository.

```
$ git clone gitlab@gitlab.cse.unsw.edu.au:COMP677/22T2/students/z5555555/ass2.git
```

(Note: Replace z5555555 with your zid)

Navigate inside the directory. You can then open vscode with `code .` (note the dot).

You may wish to setup a similar `debugging_main` function like you had in assignment 1. We will leave this to you to setup, if you feel you need it.

Running your tests

Similar to the first tutorial, you simply to have to run `Ctrl+Shift+P` and then type `Run Test` and hit enter. VS Code will compile and run all of your tests and produce an output.

Adding more tests

Part of your assignment mark will come from the quality and extensiveness of tests that you write.

You can add more test files to the `test/euclidean_vector/` directory. Simply copy `test/euclidean_vector/euclidean_vector_test1.cpp` into another file in that directory.

Note, everytime that you add a new file to the `test/euclidean_vector/` directory you will need to add another few lines to `test/CMakeLists.txt`. You can once again, simply copy the test reference for `euclidean_vector_test1.cpp` and rename the appropriate parts. Every time you update `CMakeLists.txt` in any repository, in VSCode you should codess `Ctrl+Shift+P` and run `Reload Window` for the changes to take effect.

Marking Criteria

This assignment will contribute 25% 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 algorithms instead of a dozen if statements).

50%	**Correctness** The correctness of your program will be determined automatically by tests that we will run against your program. You will not know the full sample of tests used prior to marking.
25%	**Your tests** You are required to write your own tests to ensure your program works. You will write tests in the `test/euclidean_vector` directory. At the top of each file you will also include a block comment to explain the rationale and approach you took to writing tests. Please read the Catch2 tutorial 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). At least half of the marks of this section may be awarded with the expectation that your own tests pass your own code.
20%	**C++ best practices** Your adherence to good C++ best practice in lecture. 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. We will also penalise you for standard poor practices in programming, such as having too many nested loops, poor variable naming, etc.
5%	**clang-format** In your project folder, run the following commands on all cpp/h files in the `source` and `test` directory. `$ clang-format-11 -style=file -i /path/to/file.cpp` If, for each of these files, the program outputs nothing (i.e. is linted correctly), you will receive full marks for this section (5%). A video explaining how to use clang-format can be found HERE.

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.

The following actions will result in a 0/100 mark for Euclidean Vector, 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.

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

PLEASE NOTE: We have a record of ALL previous submissions of this assignment submitted. If you find a solution from a friend, or online, we will find it and you will receive 0 for the assignment and potentially 0 for the course. Trust me, at least 1 person does it every term and I encourage you not to think you'll get lucky.

Submission

This assignment is due *Monday 11th of July, 19:59:59 (Week 7)*.

Our systems automatically record the most recent push you make to your master branch. Therefore, to "submit" your code you simply need to make sure that your master branch (on the gitlab website) is the code that you want marked for this task.

It is your responsibiltiy to ensure that your code can be successfully demonstrated on the CSE machines (e.g. vlab) from a fresh clone of your repository. Failure to ensure this may result in a loss of marks.

Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 0.2% up to 120 hours late, after which it will receive 0.

For example if an assignment you submitted with a raw awarded mark of 90% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 99%).

If the same assignment was submitted 72 hours late it would be awarded 85%, the maximum mark it can achieve at that time.

This late penalty has been amended from the original specification, and you should not assume it will be the same for future assignments.