

CSE333 Exercise 7

Out: Wednesday, February 2

Due: Wednesday, February 9 by 11 am PST

Rating: 3 (note)

Goals

- Write additional C++ classes
- Create a C++ namespace
- Write non-member friend functions to accompany C++ classes
- Use dynamic memory allocation in C++

Background

In the previous exercise, you defined a C++ class `Vector` that had a lot of functionality. In this exercise, you will modify how you store the vector data and add more functionality to your `Vector` class that should motivate you to think about C++ class design decisions.

Problem Description

Note that some of the specification is similar to Exercise 6, but there are two major changes:

1. The member data is stored in a heap-allocated array.
2. The lack of getters will change how you organize and implement your methods and non-member functions!

Create a C++ class `Vector` that implements 3-D vectors in the following two files.

- **`Vector.h`** : A header file that declares a class `Vector` with the following properties:

- The representation of a `Vector` should be an array containing three `float` s giving the magnitudes in the x, y, and z directions. The array should be dynamically allocated on the heap when a `Vector` is created and deleted when the `Vector` no longer exists.
- There should be a default (0-argument) constructor that initializes a `Vector` to $(0, 0, 0)$, a constructor with 3 `float` s as parameters giving initial values for the x, y, and z magnitudes (in that order), and a copy constructor.
- There should be a destructor that does whatever work is needed when a `Vector` object is deleted. If no work is needed, the body of the destructor can be empty.
- The class should define assignment on `Vector` s ($u = v$).
- The class should define updating assignments on vectors ($u += v$ and $u -= v$) that perform element-by-element addition or subtraction of the `Vector` components.
- Operators $+$ and $-$ should be overloaded so that $u + v$ and $u - v$ return new `Vector` s that are the sum and difference of `Vector` s u and v , respectively.
- Operator $*$ should compute the inner product (i.e., the dot product) of two `Vector` s as a `float`. If $v1 = (a, b, c)$ and $v2 = (d, e, f)$, then $v1 * v2$ should return the scalar value $a*d + b*e + c*f$.
- Operator $*$ should also be overloaded so that if v is the vector (a, b, c) and k is a `float`, then **BOTH** $v*k$ and $k*v$ should return `Vector` s containing the components of v multiplied by k (i.e., $a*k, b*k, c*k$).
- Define stream output so that $s << v$ will write `Vector` v to stream s as (a, b, c) (i.e., a left parentheses followed by the x, y, z components of v separated by commas (and no spaces), and a right parentheses).
- The `Vector` class and associated functions should be placed in a namespace `vector333`.

Note that several of these functions are required to return new `Vector` s. This means actual `Vector` values, not pointers to `Vector` s that have been allocated elsewhere.

- **Vector.cc** : A file containing the implementation of the `Vector` class.

In addition, you should create the following two files for testing and compiling your `Vector` class:

- **ex7.cc** : A file containing a `main` function that tests the `Vector` object.
- **Makefile** : The command `make` should compile the source files as needed to create an executable program named `ex7`. The command `make clean` should remove the `ex7` executable file, all `.o` files, and any editor or other backup files whose names end in `~` (e.g., `ex7.cc~`).

Implementation Notes

Getting Started

We recommend referring to: your ex6 solution, the sample ex6 solution, and examples from lecture while implementing this exercise. We also suggest you implement operators and test them one at a time. After everything has been implemented, make a style pass and double-check to see if a function should be member, non-member, or a non-member friend function.

Testing

As opposed to ex6, you do NOT need to explicitly verify the proper behavior of all of your implemented `Vector` functionality in `main`, though you should still do proper testing on your own. We strongly recommend that you write code that tests (1) `cout << v`, (2) `k*v`, and (3) `v*k`. Be sure that if an error is encountered during testing, the testing code handles it appropriately. Since there is a significant amount of functionality to test, you should write static helper functions to factor out redundant code.

Makefile

Your Makefile should contain intermediate `.o` files for each `.cc` file that will be compiled into the executable. Your Makefile should also set dependencies correctly such that it only recompiles individual files and rebuilds the program when needed, and should reuse any existing `.o` files or other files that are already up to date.

Memory Management

Make sure that you use `new` and `delete`, as opposed to their C counterparts. You should also make sure you follow the best practices for handling dynamically allocated memory in objects. In particular, you should make sure that you investigate all functions related to the rule of three. Make sure your main function has no memory leaks, even for non-typical exits (e.g., handling edge cases or test failures). We will be using `valgrind` (`valgrind --leak-check=full ./ex7`) to test your code for memory issues.

Style Focus

Don't forget that good practices from previous exercises still apply! In particular, check the ones from Exercise 6, which still apply here.

Function Access

You will have to decide what functions of the `Vector` class to make member functions and which to make non-member functions. Some can only be one or the other, while others can be implemented either way but have stylistic preferences. This can be one of the more subtle and confusing parts of C++ classes, so be sure to review the related lecture material carefully!

Friend Functions


In addition to deciding whether a function is a member or non-member function, you should decide whether a function is declared as a `friend` function or not. Be sure that you only give `friend` access to non-member functions that require access to the `private` members of the class.

Submission

You will submit: `Vector.h` , `Vector.cc` , `ex7.cc` , and `Makefile` .

Your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu` , or CSE home VM).
- Have no runtime errors, memory leaks, or memory errors (`g++` and `valgrind`).
- Be contained in the files listed above with your `Makefile` compiling your code with the `g++` options `-Wall -g -std=c++17` .
- Have a comment at the top of your `.cc` and `.h` files with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code (`cpplint.py`).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on  Gradescope (<https://www.gradescope.com>). Don't forget to add your partner if you have one.