CSE333 Exercise 6

Out: Friday, January 28
Due: Wednesday, February 2 by **11 am PST**
Rating: 4 (note)

Goals

- Write your first C++ class – data members, methods, and access specifiers
- Examine the effects of pass-by-value and pass-by-reference in C++ on function arguments and return values.
- Define the behavior of standard operators for a user-defined class.

Background

A major addition in C++ compared to C is the introduction of classes! However, there are lots of new stylistic considerations to properly encapsulate (https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)) these classes, including:

- Const and non-const variables, parameters, and return values
- Reference vs. non-reference parameters and return values
- The presence or absence of constructors, destructors, and operators
- Member vs. non-member functions (saved for Exercise 7)

In this exercise, we will put most of these pieces together while writing your first C++ class; carefully consider why each of these different pieces is important!

Problem Description

Create and test 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:
  - For this exercise, you should implement all methods as *member* functions (*i.e.*, defined within the class).

- The representation of a `Vector` should be three `float` s giving the magnitudes in the x, y, and z directions.
- The following constructors should be present:
  1. a default (*i.e.*, 0-argument) constructor that initializes a `Vector` to `(0,0,0)`,
  2. a constructor with 3 floats as parameters giving initial values for the x, y, and z magnitudes (in that order), and
  3. 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 vectors (`u = v`).
- The class should define getters `get_x` , `get_y` and `get_z` that return the corresponding floats.
- 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.
- 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` .

- **`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:

- **`Makefile`** : The command `make` should compile the source files as needed to create an executable program named `ex6` . The command `make clean` should remove the `ex6` executable file, all `.o` files, and any editor or other backup files whose names end in `~` (*e.g.*, `ex6.cc~`).

- `ex6.cc` : A file containing a `main` function that tests the `Vector` object. There are two types of tests you should do in this file:

  1. Tests that verify the correct behaviour of the defined constructors and operators of your `Vector` class. For this part of the tests, make sure you use status code to perform proper automated testing as you did in Exercise 3.

  2. After testing the behavior of the constructors and operators, test the behaviour of how objects and references are passed in C++. To do this, create two additional functions:
     - `Increment()` , which accepts an instance of a `Vector` (not a pointer or reference to a `Vector`) as an argument, and increments the values stored in the vector by one.
     - `VerifyAddress()` , which accepts a `Vector` reference and a `void*` as arguments, and returns `true` if the address of the aliased `Vector` has the same numeric value as the passed-in `void*` . It should return `false` if they are not the same integral value.

     For these tests, you should create an instance of `Vector` and then:

     1. Call `Increment()` to determine whether the `Vector` object itself was passed by value or passed by reference to `Increment()` .
        a. If it was passed by value, print out on a new line:

```
Vector: pass-by-value
```

   b. If it was passed by reference, print out on a new line:

   ```
   Vector: pass-by-reference
   ```

2. Call `VerifyAddress()` to determine whether the `main()` instance of `Vector` is the same instance as the one visible to `VerifyAddress` by comparing their addresses.
   a. If they are the same (*i.e.*, `VerifyAddress` returned `true`), print out on a new line:

   ```
   Ref: same address
   ```

   b. If `false` is returned, print out on a new line:

   ```
   Ref: different address
   ```

Note that neither `Increment()` nor `VerifyAddress()` should print the tests results; they should be printed from `main` **as the last two lines of your output**.

Implementation Notes

Memory

No dynamic allocation should be necessary for the completion of this exercise (*i.e.*, do NOT use `new`, `delete`, `malloc`, or `free`).

Testing

For the purposes of this exercise, it is enough for you to write code that makes use of each of the constructors, operators, and other functions that you wrote and verifies that the `Vector` members contain the correct values at each step. Be sure that if an error is encountered during testing, that 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.

Style Focus

Don't forget that the good practices from previous exercises still apply, however, some guidelines from C are *superseded* by C++ guidelines!

Classes

This is your first exercise with classes, and you should be sure you are following the best standards with commenting and organizing your classes as laid out in lecture. Make sure that you have comments with their declaration in the header file, and have an overall comment describing the behavior of the `Vector` class.

Const

Functions, parameters, and variables should be labeled with `const` appropriately throughout your program. Refer to the lecture material for best practices.

Submission

You will submit: `Vector.h` , `Vector.cc` , `ex6.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.