CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework 4[1]

Assigned: October 16, 2001.

Due: October 23, 2001, at 5pm.

Credit: 20 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Problem statement **1**

2 Details **3**

3 What files do I need? **3**

4 Testing **3**

5 What to submit and how **4**

6 Programming tip/rant **4**

1 Problem statement

You are to implement a class for mathematical vectors, as (briefly) discussed in class. In mathematics, a *vector* is an n-tuple of real numbers, which we could write as $(x_1, \ldots, x_n)$. n is called the dimensionality of the vector. Many operations are defined on vectors, including the following.

- Addition of two vectors of dimensionality n produces another vector of dimensionality n:

$$(x_1, \ldots, x_n) + (y_1, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n)$$

- Subtraction of two vectors of dimensionality n produces another vector of dimensionality n:

$$(x_1, \ldots, x_n) - (y_1, \ldots, y_n) = (x_1 - y_1, \ldots, x_n - y_n)$$

- Scalar multiplication of a vector of dimensionality n and a real number a produces a vector of dimensionality n:

$$a \cdot (x_1, \ldots, x_n) = (a \cdot x_1, \ldots, a \cdot x_n)$$

[1] © 2001 Jeffrey D. Oldham (oldham@cs.stanford.edu) and Berna L. Massingill (bmassing@cs.trinity.edu). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

- The *dot product* of two vectors of dimensionality n produces a real number:

$$(x_1, \ldots, x_n) \cdot (y_1, \ldots, y_n) = x_1 \cdot y_1 + \ldots + x_n \cdot y_n$$

- Two vectors are considered to be equal if they are identical, i.e., they have the same dimensionality and corresponding elements are equal.
- The mathematical length of a vector is the square root of the dot product of the vector with itself:

$$|(x_1, \ldots, x_n)| = \sqrt{x_1 \cdot x_1 + \ldots + x_n \cdot x_n}$$

The objective in this assignment is to define a C++ class `MVector` to represent mathematical vectors. Program `mvector-use.cpp`[2] illustrates the intended use of the class. Our strategy for implementing `MVector` is to represent a mathematical vector by a C++ STL `vector` of `double`s. We will need to write functions to support the following operations on `MVector` objects (using syntax as illustrated in `mvector-use.cpp`).

- Construct a zero-dimensionality `MVector`, an `MVector` of a specified dimensionality, or an `MVector` that is a copy of another `MVector`.
- Get or change the value of a specified element of an `MVector`.
- Get or change the dimensionality of an `MVector`.
- Form a new `MVector` by adding two `MVector`s.
- Form a new `MVector` by subtracting on `MVector` from another `MVector`.
- Increase the current `MVector` by adding to it another `MVector`.
- Compute the dot product of two `MVector`s.
- Form a new `MVector` as the scalar product of a `double` and an `MVector`.
- Scale the current `MVector` by a `double` a; i.e., modify the current `MVector` such that its new value is the scalar product of a and its old value.
- Compare two `MVector`s for equality and inequality.
- Input `double`s into an `MVector` using the `>>` operator. (We will use the `MVector`'s current dimensionality to determine how many elements to input.)
- Output an `MVector` using the `<<` operator.
- Compute the mathematical length of an `MVector`.

We also need to define a type `size_type`, to be used to represent the dimensionality (number of elements) of an `MVector` and indices of its elements. (Yes, we could just use `int`, but defining a new type makes it clearer that values of this type are sizes or indices, and it allows us to be more compatible with the STL `vector` class, which also defines such a type.)

[2] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW04/Problems/mvector-use.cpp`

2 Details

`mvector.h`[3] contains a partial definition of the `MVector` class; your mission is to complete this class definition. Comments in the code (`ADD SOMETHING HERE`) indicate what you need to implement. Note the following:

- Some of the desired functions or operators could be implemented in different ways (e.g., using either a friend function or a member function); you may use any method that works with the provided example-of-use program (`mvector-use.cpp`).
- For some functions, comments specifically direct you to use a particular method of implementation (e.g., an STL function). You will only receive full credit if you use the required method, although you will receive partial credit for any method that produces the desired result.
- If the comments describing a desired function say, e.g., "you may assume that the vectors are the same dimensionality", then you are not required to do any error checking to be sure that this true — although you may do such error checking if you like; see the section "Programming tip/rant" below for a suggestion.

3 What files do I need?

There is one essential file and one recommended file.

- `mvector.h`[4] contains the skeleton of the `class MVector` definition.
- `mvector-use.cpp`[5] uses the `MVector` class. You will probably want to extend this program to further test your implementation.

To test your `MVector` class definition, you will need a C++ program (a `.cpp` file) containing a main function definition and an `#include "mvector.h"`. `mvector-use.cpp` is an example of such a program; you may modify it or write your own program. Compile the `.cpp` file containing the main program; for example,

```
g++ -Wall -pedantic mvector-use.cpp -o mvector-use
```

Remember to put file `mvector.h` in the same directory as the `.cpp` file you are compiling.

4 Testing

You are encouraged to test your code thoroughly with the example-use program (`mvector-use.cpp`), extended with additional tests. As currently written, this program will not compile unless you implement all the desired `MVector` functions. If you want to implement and test functions a few at a time, you can do so by commenting out the parts of `mvector-use.cpp` that test functions you have not yet implemented.

[3] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW04/Problems/mvector.h`
[4] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW04/Problems/mvector.h`
[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW04/Problems/mvector-use.cpp`

5 What to submit and how

Submit your source code as described in the Guidelines for Programming Assignments[6]. For this assignment, use a subject header of "cs1321 hw4", and submit a single file containing your revised version of `mvector.h`. You do not need to send `mvector-use.cpp`; I will provide a more thorough test program when I test your code.

6 Programming tip/rant

It is often reasonable and convenient to write functions in which "sane" assumptions are made about function inputs; for example, in the unary-numbers class of Homework 2, we assumed that the `natural` passed to `sub1()` would not be zero. If such assumptions are made, they should be documented in comments preceding the function. Whether to check that the actual values passed to the function meet the precondition (and what to do if they don't) is at least in part a matter of programming style. For preconditions that should be checked in a calling program (but might not be, if the programmer writing the calling program is careless), I like to use the `assert()` function (`#include <cassert>`) to double-check. `assert(`*boolean_expression*`)` tests *boolean_expression*; if it is false, the program halts with an error message pointing to the call to `assert()`. Here is a throw-away program illustrating use of `assert()`:

```
#include <iostream>
#include <cassert>
int main(void) {
  assert(false);
  return 0;
}
```

When compiled and executed on a Linux system, the program produces the following output:

```
qq:  qq.cc:4:  int main():  Assertion 'false' failed.
Aborted (core dumped)
```

`mvector.h` contains additional examples of using this function.

Notice that while the error messages produced by `assert()` can be very helpful to a programmer (either the person who originally wrote the code, or another programmer using it), they are less likely to be useful to end users of application programs. Therefore, I recommend using `assert()` to check for things that "should not happen" (e.g., calling functions with parameters that don't meet the preconditions) rather than for potential user errors (e.g., the program expects the user to supply file `SomeFile`, but no such file can be found). For end-user errors, I prefer to print a message that would be meaningful to the user and bail out of the program as gracefully as possible.

For this assignment, the directions for several functions indicate that you can make assumptions about the parameters (e.g., "+" is applied only to vectors of the same dimensionality). You are not required to check that the actual parameters satisfy these assumptions, but you may if you wish. `assert()` might be a nice way of doing this.

[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`