CS 202 Summer 2022 - Assignment 4

Operator Overloading and Recursion

Overview

Unlike other assignments, this one will be more of a worksheet/ collection of functions to implement rather than a cohesive program. You can think of it as similar to a library - it is a collection of functions we provide for others to be able to use in their code.

There are two parts to this assignment, each with their own separate program. The first will focus on implementing operator overloading for a custom Vector class that we will write to behave like a math vector. This will involve writing a few math operations as well as operators to print, read into, and copy over Vectors. The second program consists of three recursive functions each solving a different problem. You can find the files needed for each part in their own folders in the handout. Each should be compiled and run separately and will be used for different tests when submitted to CodeGrade.

Recall from math that a vector is made of a series of values each representing a magnitude in a different dimension. For example, $\langle 3, -4, 0, 2\rangle$ is a 4 dimensional vector. For this program, you will need to implement a Vector class that dynamically allocates an array to store these values and keeps the size allocated. We will then implement the ability to add two Vectors using the + operator overload, a * overload to find the dot product between two vectors, and then a » and « overload to read and write the Vector from a stream.

The + overload will take two Vectors and add them elementwise to make a result Vector. For example, if u = $\langle 4, -2, 1\rangle$ and v = $\langle 1, 0, -3\rangle$, then u + v = $\langle 4+1, (-2)+0, 1+(-3)\rangle = \langle 5, -2, -2\rangle$. More formally, for two Vectors, u and v, of dimensionality k, the addition of these two is $\langle u_0 + v_0, u_1 + v_1, ..., u_{k-1} + v_{k-1}\rangle$.

The * overload for two Vector will return the dot product of those two Vectors. Recall from math that a dot product is a scalar, so a single number. It is found by taking the sum of all products between elements in two vectors. Given the same u and v from earlier: u*v = 4*1 + (-2)*0 + 1*(-3) = 1. As a formal definition, again given vectors u and v of dimensionality k:

$$\sum_{i=0}^{k-1} u_i * v_i$$

Finally, we will implement some non-math functions for the Vectors. The « operator will need to print the Vector to the given stream, while the » will read values for a Vector from some stream. We will also provide a = overload to allow us to copy the contents of one Vector to another.

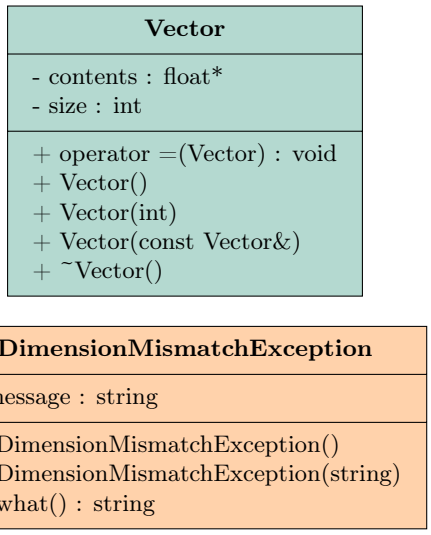
The recursion program contains three functions, each solving some small problem recursively. The first of which is a power function that simply gives a base value to a power. For example $2^4 = 16$. Try to find a base case that is trivial to find (HINT: Is there a power that always gives the same value regardless of base?). Remember that when doing recursion, we have some base case to act as a stopping point and return some value, then a recursive case that calls the function itself with different arguments to approach that base case. Make sure this function calls itself and performs some reduction on one of the arguments each time.

The next function attempts to check if a value is in an array. This function will use the binary search approach to find a number, which assumes that the array is already sorted beforehand. This should check the middle of the current array and if the element matches, we are at a base case and the value is found. Else, try to find the value in either the left or the right sub array depending on whether the target value is bigger or smaller than the middle. As an example of a single run through of this function, given the array 3, 5, 6, 8, 12 and the value 12, this should see that the middle, 6, is not the desired value and try the 8, 12

sub-array to the right, since $12 > 6$, so the value must be in the right array. See the related video for this assignment for a more detailed overview on binary search.

The last function to implement will be a function to try and find the position of a 'X' given a 2D maze array of characters. This is most easily understood by looking at existing arrays in the provided recursion.cpp. This function will take a position to check in the array and see if it can find the character 'X' there, if not, it will continue its search. To make sure we don't re-check spaces we have already checking in the array, previously checked spaces will be marked with a 'O'. If you end up in a space containing an 'O', simply return without trying adjacent spaces. Otherwise, recursively try to find the 'X' by starting at the spaces above, below, right, and left of the current one as long as they are in bounds of the array. You will need to do bounds checking using the provided WIDTH and HEIGHT members. As a hint, try separating your checks for each individual direction - so check if moving right remains in bounds before going right, check the left, etc.

UML Diagrams



Important Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, **functions implemented for you already are in blue**, and **functions that are abstract that will be implemented later are in magenta**.

Vector

The Vector class works similar to a Vector you might see in math and supports a few different operations that we will see in a later section. For convenience, the Vector class will be able to have math, printing, and assignment operations so that programmers who use our class can more cleanly utilize it.

- ***float* contents*** - A pointer to a 1D array of floats that represents the contents of our Vector. Remember that a Vector in math can be seen as a Matrix where one of the dimensions is 1 (i.e. it is a single row/column of numbers)
- ***int size*** - The size of our array. This also represents the total count of numbers in our vector
- **Vector()** - The default constructor for the vector class that simply sets the array to be unallocated and point to nullptr and the size to 0. This constructor is used by main when it wants to create a Vector using the » operator overload

- **Vector(int size)** - A constructor that is given the size of the Vector to be made. This should allocate an array of the given size and set the ***size*** class member to the passed size parameter so that we may use it later in other functions
- **Vector(const Vector& other)** - The copy constructor should make a deep copy of the given other Vector and set all of the members of the Vector being constructed using other as a template. This should set the size member to be the size of the other Vector. Then, make a deep copy of the other's contents by allocating new space to hold all of the numbers and copying them over to the allocated array
- **~Vector()** - The destructor should deallocate our dynamically allocated contents array, assuming it has been allocated.
- **void operator =(Vector other)** - The assignment operator overload should work exactly like the copy constructor. That is, it should make a deep copy of the other Vector given as a parameter and copy all of its members over to *this*object. The difference here is that = may be used after a Vector has already been constructed, so it is possible it may already have existing memory allocated. You can use the same code as the copy constructor, but make sure to add code to deallocate the contents array before making a deep copy if the contents array has already been allocated

DimensionMismatchException

The DimensionMismatchException class is a custom exception class we will use alongside the Vector class to signal that we could not do an operation. This class is already implemented for you, you will just need to use it appropriately. The main() for the vector program already contains try/catch blocks to handle exceptions that may be thrown, you will only to do the actual throwing in the event an operation is invalid.

- ***string message*** - Holds a message describing the exception that occurred
- **DimensionMismatchException()** - The default constructor simply sets the message to be blank for this exception
- **DimensionMismatchException(string message)** - Sets the message for this exception to be the passed string. This constructor should be called when creating exceptions to be thrown by the Vector operators
- **string what()** - Returns the message stored for this exception. In other words, returns some information about why exactly this exception occurred

Global Functions for Vectors

The majority of the Vector operators will be implemented globally (i.e. statically). This means they are not class members. However, we still want these functions to have access to private members, so they are marked as friends of Vector class within its body.

- **Vector operator + (Vector v1, Vector v2)** - Adds two Vectors elementwise and makes a new Vector as the result. This should go through each Vector and add corresponding elements, storing the result in a final Vector to be returned. This operation should throw a **DimensionMismatchException** if the sizes of the two Vectors do not match and cannot be added. When throwing an exception, add a message *"v1 and v2 are of different sizes and could not be added"* that describes the exception. This message is shown when **what()** is used on a caught exception
- **float operator * (Vector v1, Vector v2)** - Multiplies two vectors by finding the dot product between them. The dot product is defined as a scalar value that is the sum of the product of all corresponding elements. In other words, go through each Vector and for each element at the same index, multiply them and add them to a running sum. While this is a Vector operator, the result is just a single float. Similar to the + overload, this should throw a **DimensionMismatchException** with a relevant message if the Vectors are not of the same dimensionality and cannot be lined up properly. The message should be similar, just with "added" replaced with "mutliplied" (see output for example)

- **ostream& operator « (ostream& out, const Vector& v)** - Prints the vector in the form $\langle x_0, x_1, ...x_n \rangle$ to the given out stream, then returns the stream.
- **istream& operator » (istream& in, Vector& v)** - Reads from the given input stream into the passed Vector. This should first read the size to make the vector, then allocate an array of the size for the contents. Finally, read numbers from the stream until the whole array is filled. It is also recommended to deallocate any existing contents before allocating just in case the Vector already contains some values. This function returns the stream it read from

Recursion Functions

For the recursion portion of the assignment, we will be implementing three recursive functions. When running the main program, it will ask which one of the three functions to test, which can be selected with a number 1-3. It is encouraged to look at at main to get an idea of the input for each function. All functions are implemented globally. The descriptions for these functions is relatively brief; the goal here is to focus on problem solving

- **int power(const int& base, int power)** - This function should return the base to the given power recursively. For example power(2, 3) returns 8. Remember to consider what the trivial base case is for taking a power and to have the function call itself as part of the return value in the recursive case
- **bool isValueInArray(int arr[], int start, int length, const int& value)** - This function tells if the given value is found in the sub array starting at the start index with the given length. As an example, if we had the array 4, 5, 7, 8, the call *isValueInArray(arr, 2, 2, 7)* would search for the number 7 in the sub array starting at index 7 of length 2 (so that array 7, 8.
  This should recursively search for the value in the array by utilizing the binary search method. That is, try checking the middle value in the array. If it is not found, try the list to the left of the value if the target value is less than the middle and try the right list if the target value is greater than the middle. You can assume the list array is already sorted in advance. The middle and lengths of the left and right sub-arrays are calculated for you in the skeleton code for convenience, just focus on finding the right index to start with and how to do the recursive calls.
- **void findTheX(char grid[HEIGHT][WIDTH], const int& x, const int& y)** - Given a 2D grid of characters, this should attempt to find the 'X' char in the array starting at the given x, y position. Recall that when accessing an array, y generally comes before x. This should check the current spot for the 'X' and if not found, tries the 4 adjacent tiles above, below, left, and right of the current spot. If the 'X' is not found, mark the current spot as checked by setting the position to 'O'. If a space is already marked as an 'O', you can return without checking adjacent tiles, since that spot has already been checked. Before trying any one direction, ensure that that spot is in bounds of the 2D array with the given HEIGHT and WIDTH. Once the X is found, print where it was found similar to the sample output. This function does not return a value.

Compiling

Each program only contains one cpp file in this assignment, so no make file is provided. Simply compile **vector.cpp** for the vector program and **recursion.cpp** for the recursion program. Note that the vector program uses the provided input.txt file for testing the » overload, so please make sure the file is in the same directory as your program when running. Both programs take no command line arguments and can be run directly after compiling.

Sample Run

Vector Program

```
v1 = <5.7, 1.2, 0.4, -0.6>
v2 = <5.6, 0.8, 1.2, 0.5>
Testing = ...
Checking if v3 and v4 are the same
v3 = <1, 3, 0>
v4 = <1, 3, 0>
Testing math operations
v1 + v2 = <11.3, 2, 1.6, -0.1>
v1 * v2 = 33.06
Testing an invalid operation
Encountered an error when multiplying: v1 and v2 are of different sizes and could not be multiplied
```

Recursion power

```
2^10 = 1024
4^0 = 1
```

Recursion isValueInArray

```
Is 2 in the array? yes
Is 6 in the array? yes
Is 21 in the array? no
```

Recursion findTheX

```
---- Maze 1 ----
Found the X at (2, 3)
---- Maze 2 ----
Found the X at (1, 0)
```