Practical 5: More practice with recursion and inheritance

Due 26 Apr by 23:59 **Points** 0 **Submitting** an external tool

Practical 5: Recursion and Inheritance

General Instructions

All submissions for the practical assignments should be under version control. Review practical 1 for setting up your repository for this practical.

Submission is through Gradescope.

If you get stuck on one of the hidden test cases and really cannot resolve it yourself, please feel free to ask the practical tutors for guidance as to how to fully test your code. You will need to explain what testing you have already done (boundary cases, unit testing, etc)

We encourage you to finish your work before the practical session and take the session as consulting time.

Problem Description

Objective

This practical assignment will further test your knowledge on recursion, inheritance and polymorphism.

Testing Hint: it's easier if you test things as a small piece of code, rather than building a giant lump of code that doesn't compile or run correctly. As part of your design, you should also sketch out how you are going to build and test the code.

Problem

In this prac, you are asked to implement "map", "filter" and "reduce" using C++. If you know some functional programming languages, you should have heard of these three names. We will describe the ideas in the following sections.

When we use programming to solve tasks, often, we need to deal with lists of stuffs. For example, databases are essentially lists of data entries and strings are basically lists of characters. "map", "filter" and "reduce" allow convenient modification and aggregation of lists.

"map"

Firstly, let just look at the concept of “map”. Mathematically, given a list $L = [x_1, x_2, \ldots, x_n]$ and a function f, by mapping f onto L, we get

$$map(f, L) = [f(x_1), f(x_2), \ldots, f(x_n)]$$

“map” can be applied whenever we want to apply operations to every element of the list. For example, to convert a list of characters to upper cases, we just map *toUpper* function onto the list, where *toUpper* converts lower cases to upper cases. Given a list of books, if we want the corresponding list of authors, we just map *getAuthor* function onto the list, where *getAuthor(book)* returns the correct author. Given a list of game units, if we want to heal them all (e.g., a Mage casted an area of effect (AOE) healing ability), then we just map *heal* onto the list of game units.

Now let us get back to C++. For simplicity, for this assignment, we only consider lists where the elements are integers, and we only consider functions that map integers to integers.

Your implementation of *map* should at the very least provide the following functionalities:

- Given a list of integers, the resulting list has every value tripled. That is, $[x_1, x_2, \ldots, x_n]$ becomes $[3x_1, 3x_2, \ldots, 3x_n]$
- Given a list of integers, the resulting list has every value raised to the power of 2. That is, $[x_1, x_2, \ldots, x_n]$ becomes $[x_1^2, x_2^2, \ldots, x_n^2]$
- Given a list of integers, the resulting list has every member become the absolute value. That is, $[x_1, x_2, \ldots, x_n]$ becomes $[|x_1|, |x_2|, \ldots, |x_n|]$

In your implementation, you are required to use C++ standard vector for storing the lists. One possible way to implement the above functionalities is as follows:

- We can create a base class with name **MapGeneric**, which has a private method *int f(int)* that specifies the operation we want to map onto a list. This method is overridden later in the derived classes to deliver specific map operations. The function *f* can be declared to be pure virtual (which means that you will not include a definition for this function in the base class, and the base class can not be instantiated).
- The **MapGeneric** class also has a public method

  ```
  vector<int> MapGeneric::map(vector<int>)
  ```

  that takes a vector as the input and returns the resulting vector after mapping. You **must** use recursion to implement this map function.
- Three derived classes **MapTriple**, **MapSquare** and **MapAbsoluteValue** can be implemented and each one overrides the original f method according to different requirement. For example, MapTriple::f(x) should return 3x

“filter”

Mathematically, given a list $L = [x_1, x_2, \ldots, x_n]$ and a function $g(x)$ that returns a boolean value, if we filter L based on $g(x)$, we end up with only elements that pass the $g(x)$ test. That is, if $g(x_1) = g(x_3) = g(x_4) = true$ and $g(x_2) = g(x_5) = false$, then

$$filter\,(g, L) = [x_1, x_3, x_4]$$

"filter" can be applied whenever we want to select some elements from a list. For example, to filter out all digits from a list of characters, we just filter *isDigit()* on the list, where *isDigit()* determines whether a character is a digit or not. Given a list of books, if we want to search for books that are novels, we just filter
isNovel() from the list, where *isNovel()* returns whether the book is a novel or not. Given a list of game units, if we want to heal only the allied units, then we filter *isAllied()* and then map *heal()*

Now let us get back to C++. For simplicity, for this assignment, the function used to filter elements only returns boolean. Your implementation should at the very least provide the following functionalities:

- Given a list of integers, filter out the even values.
  That is, $[1, 2, 3, 4, 5, 6, 7]$ becomes $[1, 3, 5, 7]$
- Given a list of integers, filter out the positive values.
  That is, $[1, -1, 2, -2, 4, 7]$ becomes $[-1, -2]$
- Given a list of integers, select all two-digit positive numbers.
  That is, $[11, -1, 23, 4354, 1, 22, 4 - 22]$ becomes $[11, 23, 22]$

In your implementation, you are required to use C++ standard vector for storing the lists. One possible way to implement the above functionalities is as follows:

- We can create a base class with name **FilterGeneric**, which has a private method *bool g(int)* that specifies the operation we want to map onto a list. This method is overridden later in the derived classes to deliver specific map operations. The function *g* can be declared to be pure virtual
- The **FilterGeneric** class also has a public method

  ```
  vector<int> FilterGeneric::filter(vector<int>)
  ```

  that takes a vector as the input and returns the resulting vector after filtering. You **must** use recursion to implement this filter function.
- Three derived classes **FilterOdd**, **FilterNonPositive** and **FilterForTwoDigitPositive** can be implemented and each one overrides the original g method according to different requirement. For example, *FilterOdd::g(x)* should return true if and only if x is odd.

"reduce"

Mathematically, given a list $L = [x_1, x_2, \ldots, x_n]$ and a binary operator $\oplus$, then

$$reduce\,(\oplus, L) = ((((x_1 \oplus x_2) \oplus x_3) \oplus \ldots) \oplus x_n)$$

For example,

- if $\oplus$ is +, then $reduce\,(+, L)$ is simply the sum of all x_i, with $i = 1...n$
- if $\oplus$ is *, then $reduce\,(*, L)$ is simply the product of all x_i, with $i = 1...n$
- if $x_i \oplus x_j = \max\{x_i, x_j\}$, which means the binary operator $\oplus$ picks the larger operand as the result, then $reduce\,(\oplus, L)$ is the maximum over the list L
- if $x_i \oplus x_j = x_j$, then $reduce\,(\oplus, L)$ produces the last element of the list L

"reduce" can be applied for aggregating results as evidenced above. (It should be noted that here we are talking about a simplified version of "reduce", so it may not look too impressive.)
Now let us get back to C++. For simplicity, for this assignment, we only consider lists where the elements are integers, and we only consider binary operators that take in two integer operands and produce one integer result. For simplicity, you may assume no list has less than 2 elements. Your implementation should at the very least provide the following functionalities:

- Given a list of integers, use reduce to find out the minimum value.
- Given a list of positive integers, use reduce to find out the greatest common denominator of all the integers in the list.

In your implementation, you are required to use C++ standard vector for storing the lists. One possible way to implement the above functionalities is as follows:

- We can create a base class with name **ReduceGeneric**, which has a private method *int binaryOperator(int, int)* that specifies the operator. This method is overridden later in the derived classes to deliver specific map operations. The function *binaryOperator* can be declared to be pure virtual.
- The **ReduceGeneric** class also has a public method

  ```
  int ReduceGeneric::reduce(vector<int>)
  ```

  that takes a vector as the input and returns the result of reduce. You **must** use recursion to implement this reduce function.
- Two derived classes **ReduceMinimum** and **ReduceGCD** can be implemented and each one overrides the original binaryOperator method according to different requirement. For example, *ReduceMinimum::binaryOperator(x,y)* should return the smaller value between *x* and *y.*

Testing

The test script will compile your code using

g++ -o main.out -std=c++11 -O2 -Wall *.cpp

It is your responsibility to ensure that your code compiles on the university system. g++ has too many versions, so being able to compile on your laptop does not guarantee that it compiles on the university system. You are encouraged to debug your code on a lab computer (or use SSH).

You are asked to create a main function (main.cpp). It takes in one line of input. The input consists of exactly 20 integers each separated by a comma and space. Then in your main function:

- Construct a vector using these 20 integers. We denote the list by $L = [x_1, x_2, \ldots, x_{20}]$
- Convert the original list L to $L' = [3|x_1|, 3|x_2|, \ldots, 3|x_n|]$ using map (hint: use map twice).
- From L', select all positive two digit integers that are also odd (hint: filter twice). Let the resulting list be L''.

- Compute the minimum value and the greatest common denominator of L'' (using reduce). Output the results, separated by space (output should be “min gcd”).

Sample input: 6, -11, 53, -16, 73, 128, 105, 104, -71, -179, 102, 12, 21, -145, -99, 199,-156, -186, 43, -189

Sample output: 33 3

For your reference, L' = [18, 33, 159, 48, 219, 384, 315, 312, 213, 537, 306, 36, 63, 435,

297, 597, 468, 558, 129, 567],

L''= [33, 63].

Sample input: -5, -24, -123, -81, 200, 157, 84, 67, -83, -60, -72, 192, -25, -20, -50, -181,-70, -15, -108, -123

Sample output: 15 15

For your reference, L'= [15, 72, 369, 243, 600, 471, 252, 201, 249, 180, 216, 576, 75, 60,150, 543, 210, 45, 324, 369],

L''= [15, 75, 45].

Marking Scheme

Your submission should contain **main.cpp** and class cpp files and header files.
1 mark for complying with interface (compiles with my main.cpp file)
6 marks for tests (3 visible, 3 hidden)

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

GitHub Bitbucket

REPOSITORY

Select a repository...

BRANCH

Select a branch...

Upload Cancel