# 06. Full C++ OOP - Homework Exercises

Write C++ code for solving the tasks on the following pages.

Code should compile under the C++03 or the C++11 standard.

You can organize your code in .h and .cpp files (or only a single .cpp file, or a single .cpp file including only .h files), but you should ONLY submit them (don’t submit compiled files, like .exe, .o, .obj, etc.) and the files for each task should be in a separate folder (e.g. a folder with files for task 1, another for task 2, etc.). You can include the project file(s) from the IDE you use, if you want (E.g. the .cbp file if you use Code::Blocks).

The folder for each task should be named with the task number followed by what you feel describes the exercise in a few words.

E.g. a good name for the folder for task 2 of this homework would be:  
2.Sequence-Generator

Don’t worry about the name too much, just make sure the number is correct and that the files in the folder can be compiled when organized into a project.

Tasks 4 and 5 are related to each other (they’re basically one big task, separated into 2 tasks so that you can test the essential parts independently of the whole), which require some creativity, but will also exercise your skills for building larger projects.

If task 5 is too much for you, you are allowed to skip some of its functionality, i.e. it would be enough to implement correct input parsing and correct per-figure movements, without completely tracking the game state and limiting movements based on game state (e.g. you can skip the “if the King is threatened, the only valid moves are the ones that protect the King”).

NOTE: the class, variable and method names are given only as examples – you can use other names if you like, so long as they properly describe what they represent and they follow the task requirements for return type and parameters (if such are stated).

## Task 1

Add operators and methods to the LowestTermsFraction class from the exercises for:

* Multiplying by a number (i.e. multiplying its nominator), both a \* and a \*= variant. Try to reuse the implementation of one into the other
* Multiplying by another LowestTermsFraction, again – both a \* and a \*= variant
* Dividing by a number and dividing by another LowestTermsFraction, again – as the above
* Getting the reciprocal of a LowestTermsFraction (if the fraction is f, the reciprocal is 1 / f)

So, the code below should compile and run correctly:

|  |
| --- |
| LowestTermsFraction a, b;  cin >> a >> b;  a \*= 3;  a /= 3;  a \*= b;  a /= b;  LowestTermsFraction multBy3 = a \* 3;  LowestTermsFraction multByB = a \* b;  LowestTermsFraction divBy3 = a / 3;  LowestTermsFraction divByB = a / b;  LowestTermsFraction reciprocalA = a.getReciprocal();  cout << a << b << multBy3 << multByB << divBy3 << divByB; |

If you are unsure how one of the operations should work – check the mathematical definitions on the Internet.

## Task 2

Write a SequenceGenerator class, which has a pure-virtual method generateSequence(), contains a pointer to a dynamically allocated array of numbers (new double[…]), has a name for each of those numbers, and has a getName(int index) and a getValue(int index) method, both NOT virtual (Note: you can any other fields you need). Extend SequenceGenerator by FibonacciGenerator & SqrtGenerator.

The FibonacciGenerator should have a Fibonacci(int startFibonacciNumber, int endFibonacciNumber) constructor and generate the sequence of Fibonacci numbers from startFibonacciNumber (inclusive) to endFibonacciNumber (exclusive). The Nth Fibonacci number should have the name “Fibonacci N” (e.g. the 5th Fibonacci number should have the name “Fibonacci 5”). So, if we initialize a FibonacciGenerator(3, 6), it should generate the Fibonacci numbers from “Fibonacci 3” to “Fibonacci 5” (inclusive), i.e. the numbers 2, 3, 5 (check here for reference: <https://en.wikipedia.org/wiki/Fibonacci_number#List_of_Fibonacci_numbers>)

The SqrtGenerator should have a SqrtGenerator(int startInteger, int endInteger) and should generate the square roots of the numbers from startInteger (inclusive) to endInteger (exclusive). The square root of the number N should have the name “Sqrt(N)” (e.g. the square root of the number 49 would be named “Sqrt(49)”). So, if we initialize SqrtGenerator(4, 10), it should generate “Sqrt(4)”, “Sqrt(5)”, “Sqrt(6)”, “Sqrt(7)”, “Sqrt(8)”, “Sqrt(9)”, i.e. 2, 2.236, 2.449, 2.64575, 2.828, 3 (note: the values here are approximated for the example).

Make sure that your code has no memory problems (SequenceGenerator creates a   
new double[…], so it needs a destructor, which means it needs Rule of Three).

One way to test the if your class has memory problems would be to do something like:

|  |
| --- |
| for(int ind1 = 0; ind1 < BIG\_NUMBER; ind1++) {  for(int ind2 = 0; ind2 < BIG\_NUMBER; ind2++) {  FibonacciGenerator g(1, 100);  FibonacciGenerator copyConstructed(g);  FibonacciGenerator copyAssigned(1, 5);  copyAssigned = g;  // print copyAssigned and copyConstructed to see they are correct  // or just have a function which checks each of their values  }  } |

This code will call copy-constructors, copy-assignment operators, and destructors a lot of times, and you can track if your program’s memory usage increases (if you define a large enough BIG\_NUMBER) – if it keeps increasing, then you’ve got something wrong.

## Task 3

Write a SequencePrinter class, which has a pure-virtual print() method and a pure-virtual setSequence(const SequenceGenerator& sequence) method. Derive a SequencePrinterToString, SequencePrinterToFile, SequencePrinterToConsole class. Any class implementing setSequence(const SequenceGenerator& sequence) should change the sequence the current object work with, with the sequence passed-in from the method. Any class implementing print() should write the sequence to whatever target it prints to, the numbers in the sequence should be separated by spaces.

The SequencePrinterToString should be able to print a sequence to a string (by appending)

The SequencePrinterToFile should be able to print a sequence to a file (by overwriting the file with the sequence string equivalent). NOTE: you can use SequencePrinterToString, but NOT by inheriting it – SequencePrinterToFile isn’t a kind of SequencePrinterToString

The SequencePrinterToConsole class should be able to print a sequence to the console.

## Task 4

Write classes which represent the figures in a game of [chess](https://en.wikipedia.org/wiki/Chess) (Pawn, Knight, Bishop, Rook, Queen, King), played on the console. Each figure should have a symbol (single character which identifies the figure), a field to represent color (black/white), a position on the chessboard (e.g. G3), and the methods:

* getAvailableMoves(otherFigures) – gets a parameter which represents all other figures on the board (but doesn’t allow changing those other figures in any way – ensure that through the type of the parameter) and returns an array of positions to which this figure can make a valid move, based on its current position AND the positions of the other figures
* move(newPosition, otherFigures) – changes the position of the figure and returns true, if moving to that position is a valid move, or returns false if it isn’t a valid move
* NOTE: moving to another figure’s position would only be valid if the other figure is of the opposite color AND the current figures movement pattern covers that figure’s position

The data types you use are up to you, as well as any additional methods/fields/operators you want to use. Write a program which tests out each figure’s movements – make sure you first test the movements when there are no other figures, then with other figures of the same color, then with a mix of figures of different colors. The “tests” you do should be functions executed by the main function, instead of relying on the user to input data from the console (you could have that too, to do custom checks, but first all other tests should be executed – that way you’ll have a set of test results which will show you if you broke something while adding code)

## Task 5

Write a program which uses the figures from Task 4 and allows two players to play the game of chess on the console. Your program should do something like:

|  |
| --- |
| intialize board;  currentPlayer = white;  while(no winner) {  print board;  read player input to select figure  //optional: show positions to which selected figure can go  read player input to move figure //should check for validity  update board; //move the figure, remove any destroyed figures, etc.  change currentPlayer;  }  print end game board and winner; |

Printing the board should just be printing a matrix of 8x8. Each position of the matrix can be a single character (you can also add the column and row designations if you like). You can distinguish between white and black figures by printing all the black figures as lowercase symbols, while printing all the white figures as uppercase symbols. If you want, you can play around with changing console color, clearing the console after each move, etc. – but that will probably make your program not run on every OS. So if you do that, you should submit it as a separate version and also submit a version that doesn’t have those additional features and works on every OS.

The structure of your program is up to you, but a good approach would be to have a ChessRenderer which is inherited by a ChessConsoleVisualRenderer, which prints to the console (so that your code easily be extended, for example, by adding a ChessConsoleTextRenderer which just prints stuff like “White Knight at G3”), an InputParser which reads player input from the console, a ChessEngine which manages all the figures, handles moves and figure destruction, keeps track of the game state (*did the King die?/does the king have nowhere to move?/is the current player’s king under threat, meaning that the current player’s valid moves are restricted to defending the king?*), etc., and a Chess class which uses all of the above in the correct order to play the game.