CO3105/4105/4203/7105 Coursework 3

Released Dec 3, 2019 **Deadline Jan 6, 2020 5:00 pm**

This assignment consists of two tasks. The first task is of similar style to Coursework 2, but the second task is a more 'freestyle' task. This will be explained below.

Note: In both tasks, **you are NOT allowed to use primitive C/C++ arrays (whether static or dynamically allocated) or "raw" pointers**, except the handling of filenames and command-line arguments in Task 2. You are therefore strongly advised to use STL containers.

Task 1 (15% of the module mark)

This task assesses your knowledge on templates, operator overloading, exceptions and the use of STL containers. You are asked to complete two template classes.

- First, you need to complete the templated `IntMod<n>` class, which represents integers under *modulo arithmetic*. For a quick introduction to what modular arithmetic is, see `https://campus.cs.le.ac.uk/teaching/resources/CO3099/surgery/mod-crash-course.pdf` or `https://en.wikipedia.org/wiki/Modular_arithmetic`
  It will also be briefly explained in class.

  An object of this class represents an integer under mod n arithmetic, where n is the template argument. Note that here the template argument is not a type, but a variable (the integer n), but it works in the same way.

  This `IntMod` class supports the arithmetic operations of addition, multiplication and finding multiplicative inverses. For finding inverses, you can simply try each number one by one to see if it is an inverse. If an inverse does not exist, it should throw an `std::domain_error` exception. There are other functions such as constructors, comparisons and output to a stream. You need to complete the implementations of the functions as specified in the `IntMod.h` file.

- Then, you need to complete the template class `Poly<T>` for polynomials. A *polynomial* is an expression of the form $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_n \neq 0$. The non-negative integer n is called the *degree* of the polynomial (this n is not the same n in the previous bullet point), and the a_i's are the *coefficients*. For example, $3x^2 - 7x + 4$ is a polynomial of degree 2. A polynomial is fully represented by its list of coefficients (the variable 'x' has no meaning).

  Your template class should represent polynomials of arbitrarily large degrees. The data type of the coefficients is the template parameter T; for example the coefficients could be int, double or `IntMod` defined above. As long as T has an element named 0 and implements the +, == and << operators, your `Poly` template should work correctly. Note that the exponents (powers) are not templated; they are always integers, even when the coefficients are double or other types.

The only arithmetic operation that needs supporting is the addition of two polynomials. Polynomials are added by adding the coefficients of terms with the same exponent. For example, with `int` coefficients, $(x^2 + 2x + 3) + (4x^3 + 5x + 6) = 4x^3 + x^2 + 7x + 9$, and with `IntMod<2>` coefficients, $(x^3 + x) + (x^2 + x + 1) = x^3 + x^2 + 1$.

There are other functions such as constructors (create a polynomial given a vector of coefficients) and output to a stream. You need to complete the implementations of the functions as specified in the `Poly.h` file.

You will need to add private member variables to the classes. In the case of polynomials, you need some data structure to represent the coefficients. It is up to you to decide how to do this (what data structure to use); there are at least two equally valid ways of doing it.

You must not change the existing public interface of the classes, but you are allowed to add private or public member functions.

The `main.cpp` file shows some examples of how these two classes are used.

Task 2 (25% of the module mark)

This task assesses your knowledge in the use of STL containers, handling of files, and general programming skills and program design. You are asked to write a project allocation system for final-year undergraduate students.

The project allocation problem

Here is a description of the project allocation problem. Each teaching staff (supervisor) is supposed to supervise a certain number of students. This is their 'supervision load', which can be different for different supervisors. A supervisor proposes a number of projects. Each project has a 'multiplicity' which is the maximum number of students allowed to take the project. Again, different projects may have different multiplicities. It is assumed that the total number of projects offered by a supervisor (when taking multiplicities into account) is at least his/her load.

Students are given this list of projects. Each student must specify exactly four projects as their choices, ranked in descending order of preferences.

Based on this, an allocation of projects to students should be produced. The allocation produced must be *legal*. This means:

- Each student is allocated at most one project.
- The project allocated to a student (if any) must be one of his/her four choices. (This means it is possible that some students are not allocated projects.)
- The number of students allocated to a project does not exceed the multiplicity of that project.
- The total number of students allocated to a supervisor's projects (and hence supervised by him/her) does not exceed the load of that supervisor. (Staff can be underloaded. For simplicity, we assume a project can only be supervised by the supervisor that proposes it.)

You should also calculate a "score" of the allocation. Each student getting his/her first choice gives a score of 4, second choice a score of 3, third choice a score of 2, fourth choice a

score of 1, and nothing allocated gives a score of 0. The score of the allocation is the sum of all student scores.

The allocation produced also has to satisfy some *local optimality* requirements. This means:

- No student should be allocated a project lower on their preference list (or not allocated a project at all) when there are other projects higher on their preference list that are not 'full' (due to multiplicity or staff load).
- No two students would prefer swapping their allocated projects (i.e., both ranked the other student's project higher in their own preference list).

Intuitively, this means the allocation cannot be improved by making "local" changes that affect only one or two students without changing the allocation of the others.

Program input and output

Your program reads input data from three text files, supplied in the command-line arguments. For example:

```
./main staff.txt projects.txt students.txt
```

In all three files, the input consists of a number of lines. Each line represents a supervisor/project/student, and contains a number of fields separated by whitespace, as follows:

- Supervisor file: Each line consists of a supervisor id, followed by his/her load.
- Project file: Each line consists of a project id, then supervisor id, then multiplicity, and finally the project title.
- Student file: Each line consists of a student id, followed by 4 integers which are the 4 project choices in descending order of preference.

Supervisor and student ids are alphanumeric strings like your UoL userid. Project titles are strings that may contain spaces. All other fields are positive integers. See the webpage for some sample input files.

You can assume all these inputs are "nice": if the input file does not conform to these formats, the program behaviour is undefined. Similarly, you can assume that the files represent a legal input, e.g. supervisor ids in the project file correspond to those in the staff file, student project choice ids exist in the project file, etc. Otherwise, the program behaviour is again undefined.

The result of the allocation must be written to the file `alloc.txt`. Each line contains a student id, followed by a space, followed by the id of the allocated project (or 0 if no project is allocated to this student). There must be a line for each student. The final line contains a single integer, which is the score of the allocation.

The execution testing part of the marking will only consider the output of this file. You can print any messages to the screen if you so wish, but they will not count towards the marks. It is therefore important you produce an output file in the correct format.

All these files must be in the same directory as the executable program.

Algorithms

Other than the local optimality criteria above, there is no requirement for producing a 'good' allocation, although you are of course welcome to design and implement any algorithms that produce high-scoring allocations. One very simple algorithm that will produce a legal and locally-optimal allocation is as follows. In any arbitrary order, process each student one by one, and allocate him/her to his/her most preferred project that is not already 'full' (either because of supervision load or multiplicity).

Marking criteria and testing

For Task 1, the marking criteria is the same as in Coursework 2: 30% for testing, 60% for code inspection and 10% for coding style. Test suites and further details of marking criteria can be found on the module website. Your programs will only be tested on the departmental linux system.

Task 2 is unlike all the previous programming tasks; you have to develop all your program from scratch. No partially written .h or .cpp file will be provided. It is up to you to decide how to structure your program, what data structures to use, what classes to create (if any), or how the source code is split into different files (if you choose to do so). There is no requirement that the program must be very 'object-oriented'; you can write the whole program in one .cpp file without defining any classes, if you so wish. Even though the marking criteria will give considerations to these design choices, it does not mean that there is only one "correct" or "best" design. Remember, C++ supports a number of programming paradigms.

You can also name your file(s) in any way. You must supply a makefile so that when typing 'make' the program will be compiled into an executable file called 'main'. Some further instructions on producing makefiles will be given in class and in the website.

As usual, your program will only be tested on the departmental linux system, and this time with your makefile. There are no runnable test suites as in previous assignments/tasks, although there are a small number of sample input/output files and some auxilliary testing programs for you to use (see the webpage for details).

20% of the marks are awarded to execution testing, 70% to code inspection, and 10% to readability and coding style. This time the 70% code inspection marks have a 'rubric'-style marking criteria; the module webpage gives further explanations.

Submission instructions

As before, use the handin system at `https://campus.cs.le.ac.uk/handin`

For Task 1, submit only the files `IntMod.h` and `Poly.h`. For Task 2, you are allowed to submit any number of files.

Anonymous marking is achieved by having only the userid in the submission process. Please do not write down your name or other identifiable information in your submission.