

Roommate Matching

This project gives you practice building a program from a design. You will be provided a design for the following problem, and you must implement a C++ program based on this design.

The problem is to create a matching of roommates from a set of applicants who have submitted a roommate preference survey. The matching is a list of pairs of applicants, so that the overall pairing has a high degree of compatibility. Your program will begin by reading the survey data from an input file. Each record of the file will contain the preferences of one applicant for one or more of nine categories (named: Wild, Pious, SportsFan, Studious, Smoker, NightOwl, Vegan, Geek, Greek). If a preference is indicated, the individual must provide a numerical rating in the range 1 to 5, and also provide an indication of the importance of the category as a numerical rating in the range 1 to 10. (Hint: notice the striking similarity to the input for Homework 1!)

Finding high compatibility matching is complex, but using the importance rating makes the matching easier. We will use the *match score* of a pair of applicants to determine whether that pair will go into the matching. The match score for a pair of applicants is computed from the rating and importance scores for both applicants, and gives a measure of the total incompatibility for the pair. Suppose that one applicant gives a category a rating of r_1 and an importance of i_1 , and another applicant given a rating of r_2 and an importance of i_2 for the same category. Here's the logic of the matching score as computed for one category.

- $|r_1 - r_2|$ is the level of disagreement between the ratings of both applicants for the category
- $|r_1 - r_2| \times i_1$ is the first applicant's view of incompatibility for the category
- $|r_1 - r_2| \times i_2$ is the second applicant's view of incompatibility for the category
- $|r_1 - r_2| \times i_1 + |r_1 - r_2| \times i_2$ is the total incompatibility for the category

The match score for a pair of applicants is the sum of the total incompatibility for all categories for that pair, and will be in the range 0 to 720.

The algorithm we will use for finding matches is fairly simple (and not particularly intelligent). Beginning with the list of applicants from the input file, create all possible pairs of all applicants, and insert each pair into a list that is sorted in increasing order by the match score for the pairs. The actual matching is formed in a loop, each iteration of which removes the first pair from the list. If both applicants in the pair are unmatched, then the pair is added to the matching and the applicants are marked as matched. Any pair with at least one applicant already matched is discarded. The algorithm stops when there are one or no applicants left.

Input file description and sample:

Your program **must** read its input from a file named `survey.data` — use of another input file name will annoy the person conducting your project demonstration. The first three lines of the input file are a text header and should be ignored. Each remaining line of the input file will be a survey response record, conforming to the following format:

$$\langle \text{ID} \rangle \langle \text{tab} \rangle \langle \text{Name} \rangle \{ \langle \text{tab} \rangle \langle \text{Response Record} \rangle \}^+ \langle \text{newline} \rangle$$

where

$\langle \text{ID} \rangle$	xxx-xx-xxxx, where each x is a digit from 0-9
$\langle \text{Name} \rangle$	a sequence of 1 to 20 characters, possibly containing punctuation and spaces (but not tabs)
$\langle \text{tab} \rangle$	a single tab character
$\langle \text{newline} \rangle$	a single newline character

and a <Response Record> has the form <Category>:<Importance>:<Rating>, where

<Category> a sequence of 1 to 10 characters with no internal whitespace from the list given above
 <Importance> a positive integer in the range 1-10
 <Rating> a positive integer in the range 1-5

Note that the fields of a response record are delimited by colons (:).

You may assume that the input file will contain tabs and colons as described. Do not assume that all the category labels that occur in the input file will be from the list given above (case sensitive); if they are not, your program should ignore that response record. You also may not assume that the integer values will be in the specified ranges; if your program encounters an out-of-range integer value, the program should ignore that response record. Each individual will always have at least one valid response record. If no response is given for a category, then assume the applicant did not care about that category, and treat both the importance and rating for that category as zero.

```
Roommate Preferences Survey Data
Sequence Number: 4390120

000-00-0001 Hokie, Joe Pious:1:1 SportsFan:5:5 Wild:5:1 Smoker:5:1 Geek:10:5
000-00-0002 Hoo, Haskell Wild:10:5 Smoker:5:5 Greek:10:5 Geek:10:1 SportsFan:5:5
000-00-0003 Howard, Hal Pious:10:1 Wild:10:5 Geek:10:5
000-00-0004 Hopcroft, Al Geek:10:5 SportsFan:1:5 Smoker:10:1
000-00-0005 Hubbard, Bill Geek:1:1 Pious:1:1 NightOwl:10:1 Vegan:1:1 Greek:1:1
000-00-0006 Hipman, Hap Wild:10:5 NightOwl:10:5 Smoker:10:5
000-00-0007 Hom, Harry Pious:5:3 Smoker:5:2 Vegan:8:3
```

There will be at least one survey response record; the maximum number of input lines is unspecified. Your program must be written so that it will detect when it's out of input and terminate correctly.

Output description and sample:

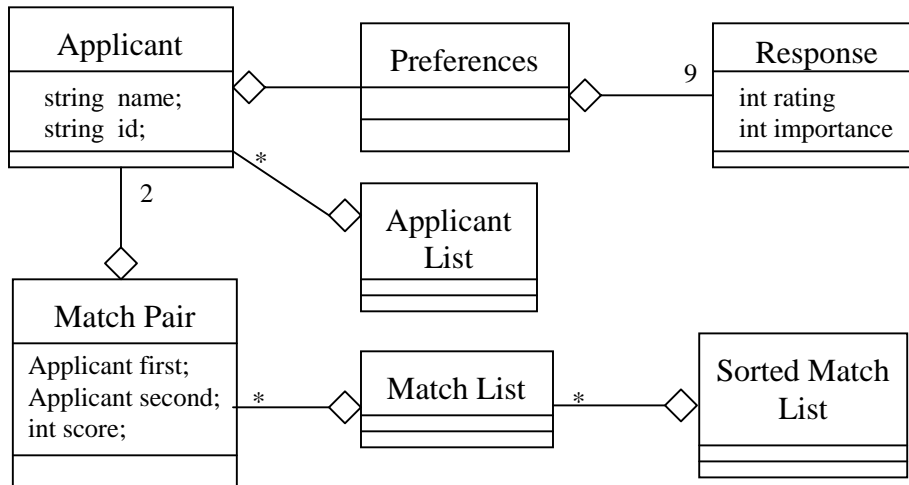
The output for the program is to be formatted as illustrated in the following example. The format does not have to be exactly identical, but should be neat, and readable.

```
Roommate Match Results
Sequence Number: 4390120

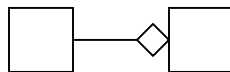
-----
Names: Hokie, Joe
       Hopcroft, Al
Score: 6
Best: Geek, Greek, NightOwl, Smoker, SportsFan, Studios, Vegan
Worst: Wild
-----
Names: Hubbard, Bill
       Hom, Harry
Score: 52
Best: SportsFan, Studios, Wild
Worst: Vegan
-----
Names: Hoo, Haskell
       Hipman, Hap
Score: 135
Best: Pious, Smoker, Studios, Vegan, Wild
Worst: Greek, NightOwl
-----
Howard, Hal *no match*
```

Design:

For this project you are to use the design given in the class diagram given below. Not all design details are given and you will need to make your own choices in determining these. In particular, you will need to decide on methods for all of the classes. (You should get in the habit of developing class and operation forms for all classes, since they will be required for future assignments.) The idea behind the design is that we want to be able to work with pairs of applicants and their ratings for the nine categories. Several times in the algorithm given in the description above we need to store either applicants or pairs in lists, so we have classes for lists. All of the class relationships in this example are aggregations, meaning that one class contains the other.



Legend:



Aggregation - object on left contained in object on right

Classes:

- Applicant – each object represents an applicant, and stores the name and identification number (as strings). You should define accessor, and mutator methods for this class, and encapsulate the data.
- Applicant List – a list of applicant objects. The aggregation in the diagram shows that zero, or more applicants may occur in the list. Details about how this list and the others are to be implemented are given below.
- Preferences – each applicant also has a set of responses for the nine categories. The nine on the aggregation shows that every Preference object should hold exactly nine Response objects.
- Response – the rating and importance responses for each category are stored as a Response object. (You may want to consider storing the category name with the Response, or possibly in the Preferences object.)
- Match pair – this class defines a pair for the matching. Each pair object has at least two applicants, and their match score.
- Match list – an unsorted list of match pairs. See discussion of lists below.
- Sorted match list – a sorted list of matches. This is a “wrapper” class that should be implemented by using the match list class.

Lists

Your list classes must be implemented using a linked list of your own construction (you should do this from scratch, rather than use code that you wrote from a previous course – it’s good for you – really). Both the applicant list and the match list should have nearly identical interfaces that allow you to insert an object anywhere in the list. Since the sorted match list contains an unsorted match list, the insert routine will insert the matches in increasing order of score. So, the insert for the sorted match list class will not insert in arbitrary positions. Example class and operation forms for generic sorted and unsorted list classes will be posted on the course website.

Deliverables:

Your final project submission must include the following:

- All source code (`*.cpp` and `*.h` files) comprising your project.
- MS Visual C++ project files (`dsp` and `dsw`) or Unix `makefile`, as appropriate. (**VC++ users:** do not submit unnecessary files, such as `ncb`, `opt`, `ilk`, `obj`, `pch`, or `pdb` files).
- One set of input/output files from your testing.
- A brief ASCII text readme file, named `readme.txt`, containing:
 - Your name and e-mail address.
 - Your instructor's name, and the section of the course you're enrolled in (example: 8:00 MWF).
 - The project name.
 - The platform and compiler you're using (example: MSVC++ 6.0 under NT).
 - Any special execution instructions. (If your program is not fully functional, be sure to mention that here and to follow the directions in the Evaluation section above.)
- An ASCII text file, named `pledge.txt`, containing the Honor Code Pledge listed at the end of this file.

Your project submission will consist of a zipped archive file containing all of the items specified above.

You are allowed to submit your solution up to five times, in case you detect (or solve) problems after your first submission. Your last submission will be the only one tested and graded. Note that, due to late penalties, it is possible that a fixed but late submission may receive a lower score than a faulty but on-time submission.

You will submit this assignment to the Curator System (as for Homework 1), and you will demonstrate its operation to the GTA of your section. In order to receive any credit for this assignment, you are required to meet with the GTA, even if your program does not compile and run.

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

No special setup is needed on your machine in order to make submissions.

Evaluation:

Your score on this project may be based on several factors:

- Runtime testing of your program, and correctness and completeness of output.
- Adherence of implementation to the supplied design, and adherence to good software engineering practice.
- Quality of internal documentation.

The relative weighting of these factors will be decided later. To receive partial credit for programs that are non-working, or are not fully functional, a brief one or two paragraph description of the problem(s) must be included in the assignment submission, in an ASCII text file named `problems.txt`. The location of the problem, minimally identified to a specific function or functions, must also be specified along with possible corrections that need to be made.

Programming Standards:

We will be evaluating your source code on this assignment for the quality of internal documentation and programming style, so you should observe good practice. Here's a brief description of some of the things we'd normally expect:

Documentation:

- You must include the honor pledge (below) in your program header comment.
- Your header comment must describe what your program does.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose.
- Precede every function you write with a header comment. This should explain in one sentence what the function does, then describe the logical purpose of each parameter (if any), describe the return value (if any), and state reasonable pre- and post-conditions.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable.

Coding:

- Use named constants instead of variables where appropriate.
- Use an struct or class variables to organize related heterogeneous data.
- Make good use of user-defined functions in your design and implementation. The body of `main()` should contain no more than 20 executable statements and the bodies of the other functions you write should each contain no more than 40 executable statements. An executable statement is any statement other than a constant or variable declaration, function prototype or comment. Blank lines do not count.
- The definition of `main()` must be the first function definition in your source file. You may use file-scoped function prototypes and you may use file-scoped constants. You may also make the `typedef` statement for your structured variable type file-scoped (in fact you must do this).
- You may not use file-scoped variables of any kind.
- Function parameters should be passed appropriately. Use pass-by-reference only when the called function needs to modify the parameter. Pass any large objects, such as array parameters, by constant reference (using `const`) when pass-by-reference is not needed.
- Use string objects to store character data, not `char` arrays.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header preceding your `main()` function:

```
// On my honor:
//
// - I have not discussed the C++ language code in my program with
//   anyone other than my instructor or the teaching assistants
//   assigned to this course.
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - If any C++ language code or documentation used in my program
//   was obtained from another source, such as a text book or course
//   notes, that has been clearly noted with a proper citation in
//   the comments of my program.
//
// - I have not designed this program in such a way as to defeat or
//   interfere with the normal operation of the Automated Grader.
```

Failure to include this pledge in a submission is a violation of the Honor Code.