School Management And Reporting Tool

Due Tuesday, March 9 at 8 a.m.

CSE 1325 - Sprint 2021 - Homework #5 - 1 - Rev 0

Elementary and secondary schools (grades 1 through 12) are organized in many cultures to transfer basic knowledge such as reading, writing, and arithmetic to young students. In the US, we usually classify grades 1-6 as elementary school, grades 7-8 as junior high school, and grades 9-12 as high school, although this varies. While public schools are usually fully funded by local and state governments, many religious, secular, and charter schools are also available.

Keeping track of students and their parents, teachers, course subjects, sections, and grades, and the many relationships between them, is of course one key to ensuring that knowledge transfer and preparation for university, vocational training, and careers is efficiently completed.

The School Management And Reporting Tool (SMART) will assist administrators of elementary and secondary schools with such tasks.

Assignment Background

School districts are constantly seeking more efficient tools for managing the plethora of data involved in school adminstration. Companies seeking to fulfill these needs (for a modest honorarium, of course) often subcontract the design and implementation of such software. One such company, Nanofruit Infinitysoft, seeks a School Management And Reporting Tool (SMART) to support school districts operating or soon to operate in various cities, states, countries, continents, and (if Elon Musk's SpaceX is successful) planets.

Your goal is to prove that you can implement their SMART (and possibly smart) specification and thus win the contract to build the full production version along with the associated fame and cash and possible spacecraft tickets for Mars on-site support.

This will be a 5-sprint, 5-week project that you can add to your growing resume, with emphasis on writing graphical user interfaces in gtkmm, file I/O, and polymorphism. (We'll have a final, stand-alone 1-week assignment to practice threading, which would be too ambitious to include in this project.) Implementation guidance will be provided each week to help you with your implementation, and you may have a few checkpoints where you may switch to the professor's suggested solution if you lose your footing.

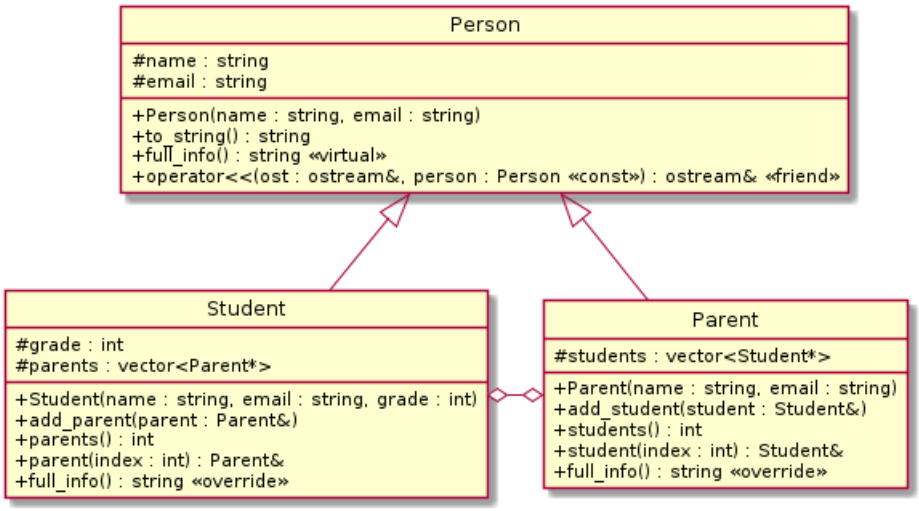
Your implementation is permitted to vary from any class diagram provided with the final project as needed without penalty, as long as you correctly implement both the letter of and the intent of the feature list. In addition, if you prefer to build a tool that addresses a different cultural approach to educating children, that would simply be delightful. **If you have questions about the acceptability of changes that you are considering, contact the professor first!**

The class diagram for the first sprint is on page 2. This consists of 3 classes, although Student and Parent are very similar (copy, paste, and global search and replace are your friend here).

For the first sprint, a supplied program will be used to test your 3 classes. Writing a command line interface program is an optional Bonus opportunity (up to 15 points). We'll begin replacing it with a graphical user interface (GUI) in sprint 2. We'll add saving and loading the data in sprint 3. The remaining sprints will continue to build out additional functionality.

Some features on the complete Scrum spreadsheet (coming soon) are not assigned to one of the 5 sprints, but have Bonus points assigned instead. After you have implemented ALL of the assigned, required features, you may implement one or more of the Bonus features for additional points to be awarded in sprint 5 only.

Class Overview



Person

Product is the base class for all students, parents, teachers, administrators, and other human participants in the SMART system. While we could track endless information, we rely here on two attributes, their name and an email address, the latter of which we expect to be unique for each person.

In addition to the constructor, which (for Sprint 1) permanently defines the attributes, we have a to_string method that returns the name and a full_info method that returns the name and (in parentheses) the email address. full_info is virtual, and is overridden in the derived classes to provide additional role-specific information.

Overloaded operator<< simply streams out the name. For Sprint 1, this operator<< will automatically apply to derived classes as well.

Two roles are provided as derived classes, Student and Parent. Additional roles will be added in later sprints. (In a real system, we probably wouldn't use inheritance for this. Indeed, we'd be using a database. But we use inheritance to give you practice in the technology.)

Student

A Student is a young Person attending classes at a SMART-managed school. SMART tracks to which grade (1 through 12) the student is assigned, which is also set by the constructor and (in the current sprint) doesn't change.

The Student class also maintains a vector of pointers to one or more Parents or other guardians for the student. Each parent is added after construction via the add_parent method.

Parents may also be accessed. The parents() method returns the number of parents identified for a student (that is, the size of the parents vector), and parent(int) will return the parent specified in the parameter. If the parameter is not within range, throw a std::out_of_range exception (it is not merely coincidental that std::vector::at does exactly this with no extra effort on your part).

Override full_info to return a string containing not only what the base class provides (hint, hint), but also the student's grade and parent names.

Parent

A Parent is an older person who has some level of responsibility for one or more students in a SMART-managed school. It is rather similar to Student (sans grade). May I again recommend copy, paste, and global search and replace?

Handling Circular Includes

Notice in the class diagram that Parent aggregates Student, and Student aggregates Parent. What could possibly go wrong?

Well, *much*, and that's quite intentional on my part, too. In C++, class Parent must refer to class Student in parent.h, while class Student must refer to class Parent in student.h. Give this a whirl in the way you've been taught in order to really understand the problem.

There, see? How can you declare a Parent, if to do so you must first declare a Student that also declares a Parent? Quite the conundrum, isn't it, like a dog chasing its own tail?

Remember, however, that *you may declare a name as often as you like*. The solution, then, is three-fold.

1. Do **not** include student.h in parent.h, nor parent.h in student.h. That won't work.
2. Instead, literally declare `class Student;` at the top of parent.h and `class Parent;` at the top of student.h. That's the minimum necessary to include each type in the class declaration of the other.
3. Finally, include student.h in parent.**cpp**, and parent.h in student.**cpp**. These *complete* declarations will then permit you to reference the class members you need to implement each class' members.

Constructor Delegation

As you hopefully learned *before* the exam, a derived class cannot construct an attribute of the base class, even if that attribute is protected. Instead, you must *delegate* to the base class constructor in the derived class' initialization list.

In addition, I might as well mention that the compiler will write the delegated versions of all base class constructors for you if you ask nicely. In your derived class declaration, simply add the statement `using Base::Base` (where Base is the base class name). You may add additional derived class constructors with different parameter lists if you like.

Test Code

I will post several test programs that you may use to verify that your classes are working correctly to cse1325-prof/P05. I encourage you to use these programs to verify that your code is correct.

Makefile

Write your Makefile with at least the following targets:

- person.o, student.o, parent.o, and main.o - compile the corresponding .cpp file including -c and $(CXXFLAGS)
- smart - link the above .o files into an executable named `smart`.
- debug - *rebuild* all .o files and the smart executable with -g as an additional parameter in $(CXXFLAGS)
- clean - force remove (at least) all .o and .gch files as well as the smart executable.

User Interface (BONUS!)

A command line interface is NOT required for this sprint. However, should you have time to write one, it will be worth up to 15 bonus points on this sprint.

Put the comand line interface code into file cli.cpp, and update your Makefile to build it into executable `cli`. Do this first, as the grader won't evaulate your code without Makefile support.

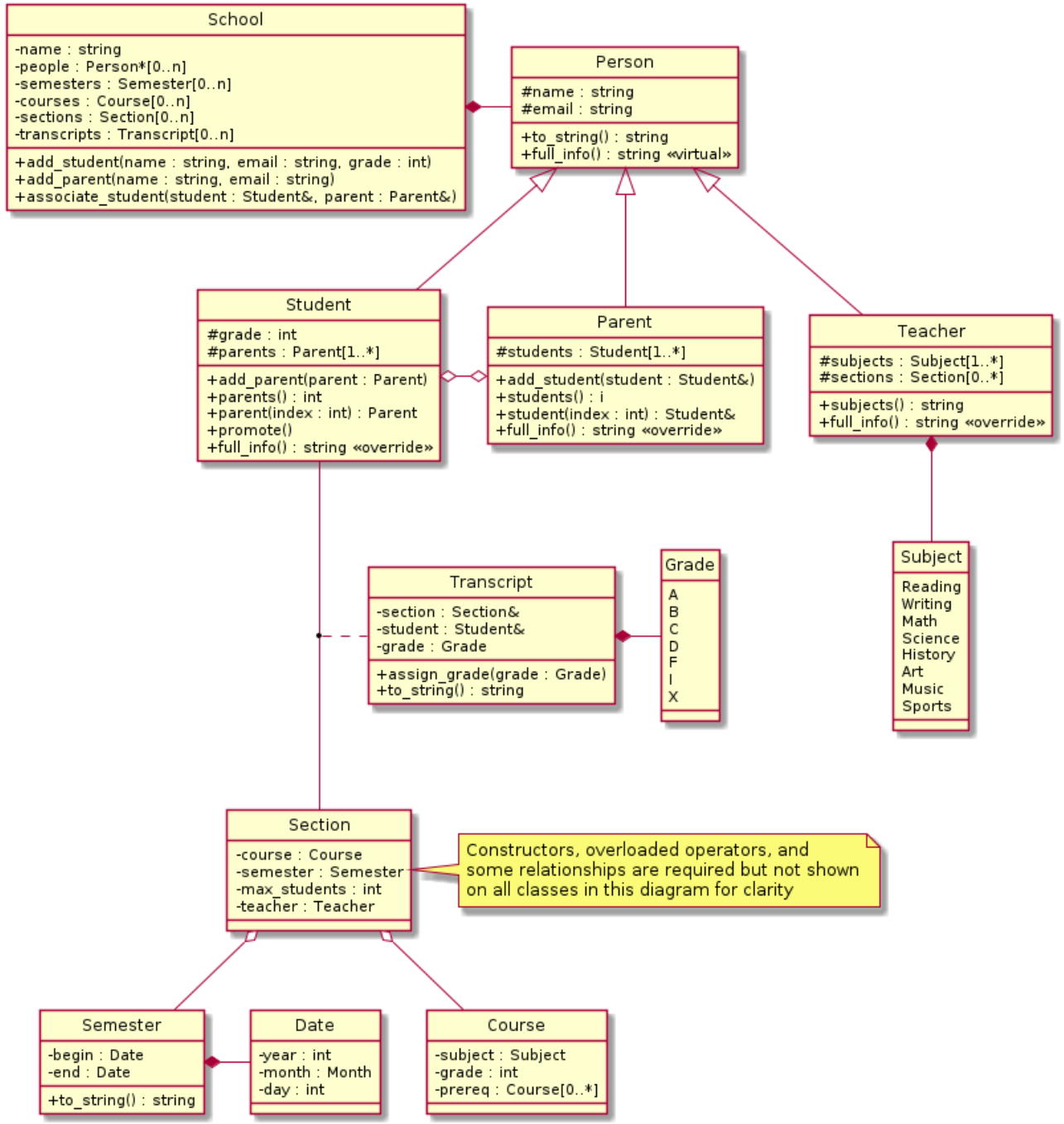
The command line interface must either include a vector of pointers to Person, or separate vectors of Parents and Students. The user interface must offer the following features:

- Create and add a new Parent
- Create and add a new Student
- Pair an existing student with an existing parent. Have the user select a Parent and a Student. Then, call Parent::add_student with the Student object as parameter, and then call Student::add_parent with the Parent object.
- List the full_info for every Student in the vector.
- List the full_info for every Parent in the vector.

As always, partial credit is awarded if some but not all of these features are completed.

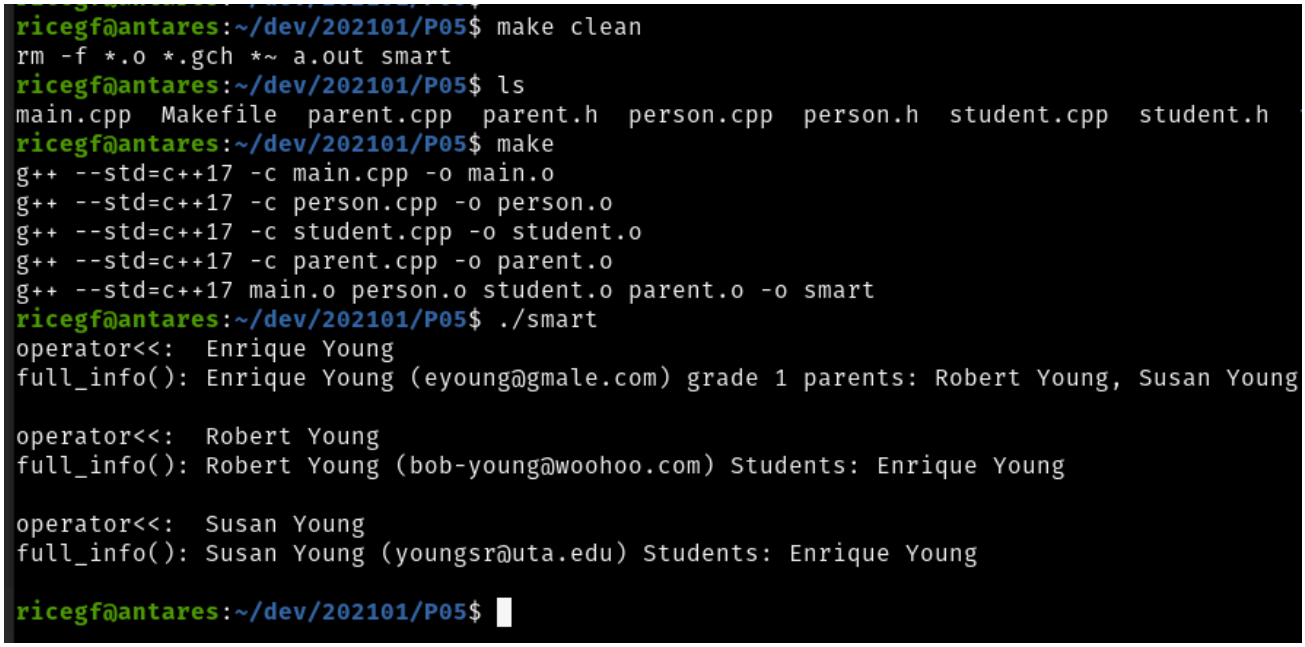
Future Sprints

The current and rather early state of the final project design, including some bonus features, is represented by the class diagram below. **This diagram is for reference only** should you decide to work ahead on features, and is *very* likely to change as the project proceeds.



Note that rather than vector<Person>, this diagram uses the standard notation Person[0..n] or similar. They are equivalent.

Here is an example run for the program. Note that the behavior of test code supplied on cse1325-prof/P05 may vary.



If you have questions about any part of this project, contact me via email or Teams. Ask questions. I **love** questions!