Lab 9: Structures and Data Abstraction

Overview

This exercise is a prelude to the design of classes. C++ structures can be used, with discipline, to design abstract data types. They do not provide the protections that classes do, but they are a good means to warm up to classes. The exercise also gives you more practice creating a multiple-file program whose parts can be compiled separately, and still more practice with file I/O.

In this lab you will define and implement an abstract data type that represents a student record. It will consist of a structure and associated functions for manipulating the data in the student record.

Review of Data Abstraction and Abstract Data Types

Procedural abstraction is the separation of what a function does from how it does it. The idea is to write descriptions of what functions do without actually writing the functions, and separate the *what* from the *how*. The client software, i.e., the software calling the function, only needs to know the parameters to pass to it, the return value, and what the function does; it should not know how the function works. In this sense, functions become like black boxes that perform tasks.

Data abstraction separates what can be done to data from how it is actually done. It focuses on the operations of data, not on the implementation of the operations. For example, in data abstraction, you might specify that a set of numbers provides functions to find the largest or smallest values, or the average value, but never to display all of the data in the set. It might also provide a function that answers yes or no to queries such as, "is this number present in the set?" In data abstraction, data and the operations that act on it form an entity, an object, inseparable from each other, and the implementation of these operations is hidden from the client.

Information hiding takes data abstraction one step further. Not only are the implementations of the operations hidden within the module, but the data itself can be hidden. The client software does not know the form of the data inside the black box, so clients cannot tamper with the hidden data.

An **abstract data type** is a representation of an object. It is a collection of data and a set of operations that act on the data. An ADT's operations can be used without knowing their implementations or how data is stored, as long as the interface to the ADT is precisely specified.

There are two views of an abstract data type: its **public view**, called its **interface**, and its **private view**, called its **implementation**. The parts of a class that are in its public view are said to be *exposed by the class*.

Example

A soft drink vending machine is a good analogy. From a consumer's perspective, a vending machine contains soft drinks that can be selected and dispensed when the appropriate money has been inserted and the appropriate buttons pressed. The consumer sees its interface alone. Inputs are money and button-presses. The vending machine outputs soft drinks and change. The user does not need to know how a vending machine works, only what its abstract data type is.

The vending machine company's support staff need to know the vending machines internal workings, its data structure. The person that restocks the machine has to know the internal structure, i.e., where the spring water goes, where the sodas go, and so on. That person is like the programmer who implements the ADT with data structures and other programming constructs.

The ADT is sometimes characterized as a wall with slits in it for data to pass in and out. The wall prevents outside users from accessing the internal implementation, but allows them to request services of the ADT.

ADT operations can be broadly classified into the following types:

- operations that add new data to the ADT's data collection,
- operations that remove data from the ADT's data collection,
- operations that modify the data in the ADT's collection,
- operations to query the ADT about the data in the collection.

The `student` ADT

The student abstract data type contains the student's last name, first name, id (9 digit number with possible leading zeros), GPA (a real number in the range of 0.0 to 4.0, a whole non-negative number of credits completed to date, and a single letter status indicating full time ('F') or part time ('P').

The functions that the ADT should make available to client software (i.e., should expose) are:

1. Given a first name, last name, id, and status, create a new student record with those values, setting the GPA and number of completed credits to zero.
2. Given a student record, return the student's id.
3. Given a student record, return the student's last name.
4. Given a student record, return the student's status.
5. Given a student record and a new status, change the student's status.
6. Given a student record and a number of credits, change number of completed credits for a student by adding the new number to the current number of credits.
7. Given a student record and a GPA, change the student's GPA to the new value.
8. Given a student record and an output stream, print the members of the record on that output stream in the format:

   ```
   lastname  firstname  id  status  GPA  credits_earned
   ```

   separated by tab characters (`'\t'`).

Exercise

You will write a small program that implements the `student` ADT described above. Specifically, you will first design a header file that contains a structure definition that represents the student data, as well as the prototypes of all functions described above. Each function prototype must be thoroughly documented. Name this header file `student.h`.

You will then create an implementation file named `student.cpp` in which you implement each of the above eight functions.

This exercise is a bit unusual because the main program does not do anything particularly logical. It exists just to exercise your student ADT. Specifically, it must do the following:

- The main program must read a file containing at most ten lines of the form

  ```
  lastname  firstname  id  status
  ```

  and for each line, create a student record with that data and a GPA and total credits earned of 0. These records should be elements of an array of student records. For simplicity assume that the file is named `studentdata`. You may hard-code its name into the main program.

- After the data is read in, the main program must display on the screen the last name and the id of every student record.

- It must then display the last name of every student whose status is 'F'.
- The program must then change the GPA of every student to 4.0, the credits earned to 16, and the status to 'P'.
- Finally, the main program must write all of the records to a file named `newstudentrecords`.

Remember that the main program is not allowed to access the data members of the structure. It must do everything only by calling the functions whose prototypes are in your header file. Name the main program `lab09_main.cpp`.
In short, the program must consist of exactly three files:

- A header file named `student.h`
- An implementation file for this header file, named `student.cpp`
- A main program file named `lab09_main.cpp`

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab09/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your three files, which should be named as described above, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with your name and other appropriate information in it.
3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab09/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document. There are absolutely no extensions to the deadline.