Detailed instructions for submitting your work, and our CSI310 project late project grading policy are provided in a separate document.

Briefly, CSI310 projects require

1. Immediate study of the project assignment and survey of its data structure topics in order to make a project management plan. Some work must begin immediately so the project is completed comfortably throughout the assignment period.

2. Development on command-line interface Unix.

3. Separate interface (.h) and implementation files (.cxx), with the implementations of the main module and each class compiled separately.

4. Documentation written in each interface file in the form of pre- and post-conditions for each method, plus representation invariants for the data members.

5. Revision tracking using RCS.

6. Scripting of the software build.

7. Electronic submission for grading. The submission project name is Proj1 for this project.

The programs you write must be organized as illustrated throughout Main and Savage’s DSO textbook. The relevant examples and explanations for this project are in Chapter 3. To see the code all in one place, use the Web to access files in http://www.cs.colorado.edu/~main/chapter3. However, it is acceptable to ignore and omit DSO’s use of namespaces (except for std::), const parameters and member functions, and typedefs in defining container classes. But include macro guards, separate header and implementation files, and documentation in the form of pre- and post-conditions (and nice looking indentation) are absolutely required. Some students wasted many hours on past CSI310 projects because of omitted and especially incorrect include guards!

Technology Training: Separate compilations were covered in Training 1. Practice the directions from pages 7-8 there so you can use a build script for Project 1 from the beginning. (Build scripts will be graded in Project 1, so no separate grading will be done for the training.) Instructions for RCS appear in this handout. Start using them right away!

Discussion Lab 3

The topics for the Lab Discussions of the Feb. 7 week are (1) the details for each these items above together with (2) creating a Coin class together with a test driver, to begin the project.

The lab would be a good place for you to try out RCS. Detailed instructions are given later in this handout.
Some discussion might be given for the Pile class, and some lab students might do a partial implementation of that.

The information about Coins, Piles and the “teller” application is written in the project description below.

You will use your Coin class for the element type in a container class called Pile to model one pile of coins which could be in a teller’s cash drawer.

**Project 1**

In this project you should end up writing software that simulates a cash drawer and a teller or cashier’s use of it. However, the educational goals are (1) to begin using some general development and object oriented design and C++ coding practices covered in Chapters 1-3 of DSO; (2) to learn properties and implementation of the partially filled array data structure presented in Chapter 3 of DSO, and (3) to solve some algorithm design and coding problems involving single and multiple arrays.

US currency has 6 denominations of coins: $0.01, $0.05, $0.10, $0.25, $0.50 and $1.00.

Each Coin has a Year and a MintMark. The Year will be input and output in plain decimal. The MintMark is a single letter.

We capitalized “Coin” because we want you to use a concrete class named class Coin for the type of data to simulate one genuine coin. The class Coin should be designed, coded and used in the fashion of the throttle and point classes from Chapter 2 of DSO.

The “Pile” class will be a partially filled array container class explained in Chapter 3; each Pile will contain a sequence of Coins.

Two parts resulting in two different executable programs are required in the project: First, the class Pile and a test driver named “testPile” for testing it must be put together.

Second, the simulator named “teller” described below must be completed and it must use class Pile developed for the first part.

The functional correctness and use of assigned data structures and algorithms in each part counts for 35% of the project grade, for a total of 70%. The remaining 30% is allotted to style, structure, use of RCS and invariants, etc.

The cash drawer has 6 separate Piles of Coins, one Pile for each denomination. (The internal structure of the teller program must therefore have 6 instances of class Pile.) The Coins in each Pile will generally be kept in “stack” order. In other words, when a Coin is put in its Pile, that Coin is always put on top; and Coins are usually removed from the top. Real money cannot be (legally) copied. So, after a Coin is removed from its Pile, it is no longer in that Pile. The number of Coins in that Pile therefore decreases by one, and those remaining Coins are left in place unmodified.

Each time the teller program is run, all 6 Piles start out empty. Your program should then print a usage menu and accept commands one at a time. For each command it should perform the requested operation if the command and data have no errors, modify the cash drawer contents appropriately, and print the results we specify in detail below.
Most user input errors should be detected and tolerated. A useful explanation of the error should be printed, the stored data not modified, and the program should continue on to the next command.

The exception is for inputting numbers...just use `cin >> VAR`. (This pattern can be used to input the mint mark, with `VAR` is of type `char`.

For the `teller` program (part 2 of the project), the operations and single letter commands are:

- **(A)** Add a coin with a given denomination, year and mint mark the cash drawer (except if the proper Pile is full).

  This operation should reject Coins that do not have legal US denominations or do not have a year between 1776 and 2006 inclusively. When a Coin is rejected, the program should not put it in any pile; it should print an informative message and then continue to accept commands.

- **(R)** Report the drawer contents, printing the denominations and all the other information for all Coins in a specified order.

- **(Y)** Yank out the newest (greatest year) Coin nearest the bottom of each Pile and put it back on top of that Pile. Therefore, all the Coins that were above the yanked Coin will move down one position.

- **(Z)** Sort each Pile so that the newer a coin is, the closer it is to the top. For Coins with the same year, make the one with the smaller mint mark in ASCII character order be closer to the top. (Notice that the newer a Coin, the greater is its year!)

  (The purpose of Y and Z is to model what the cashier might do when he/she is bored, or wish to collect old coins.)

- **(D)** Try to disburse a given (positive integer) amount of cents with only Coins currently in the drawer.

  The algorithm design problems increase in difficulty going down the list. (Hints for disburse: choose the coins going from larger to smaller denominations; imagine yourself doing it by hand.) That is not too hard except that sometimes disbursing one or more quarters ($0.25) prevents disbursing the remaining amount from the lower denomination coins. The simplest example of this situation is when the drawer only contains one quarter and 3 dimes, and $0.30 is requested. (Since the drawer only contains these 4 coins, it doesn’t contain any pennies or nickels. Stop NOW and figure out what happens when the cashier follows the algorithm and gives out the quarter. What happens next?)

  The simple disbursing, and disbursing that solves the “$0.30-problem” are programming challenges. Solutions will probably require you to design and use additional arrays or container classes, and perhaps add new features to class `Pile`.

  It is recommended that you skip these problems and first complete the minimum selection and sorting operations. If your program meets all the requirements except for disbursement with the “$0.30-problem,” it will get a score of about 90%.

  Executable reference implementations for you to try are available on the ITSUNIX file system under the names `~acsi310/Proj1/test_Pile` and `~acsi310/Proj1/teller`. 

3
To "specify a class" (for this course CSI310) means to write public member function prototypes with the required pre- and post-condition documentation, as DSO illustrated throughout Chapters 2 and 3; see DSO Chapter 3 programming projects “8” and “13” for more examples. For this project, I published the header file for you to use. Future projects will require you to write header files like this (documented with representation invariants for data and pre and post conditions for all member functions) by yourself.

The first part must provide a test driver program for class Pile fashioned after the program featured in section 3.3 “Interactive Test Programs” in DSO. The “do” statement in the main() function performs a “command detection and dispatch” loop. The user interface for the teller program can be implemented in the same way.

1 Part 1, Pile class

To begin, create a new directory named Project1 for this project and this project only. “Go to” this directory using the cd command to begin the programming work.

Study and copy from `~acsi310/Proj1/Pile.h` the file whose contents are below. Do the same for `~acsi310/Proj1/Coin.h`. Then write the test driver. “Open” and write into the Coin implementation file Coin.cxx the Coin functions that are not defined inline (inside the class definition in Coin.h). Next begin the implementation file Pile.cxx with stubs for any function body you cannot code immediately. Write a build.sh script (see Lab1). Compile the programs, correct and fix the almost inevitable syntax errors one at a time. (Rerun your build script after each single correction because compilers like g++ often don’t report accurately errors in the code that comes after the first error.)

Finally, fill in any stubs, and test/debug as necessary.

1.1 Header file Pile.h

```cpp
#ifndef Pile_h_included
#define Pile_h_included

#include "Coin.h"
#include <iostream>
#include <cstdlib>

class Pile {

public:

    Pile() {reset();} //Pre:
    //Post: The pile is empty.

    void reset(); //Pre:

```
//Post: The pile is empty.

void sort();
//Pre:
//Post: The pile is sorted from larger to smaller years going down
//the pile; for equal years, the sorting is smaller to larger
//mint marks going down the pile.

Coin take();
// Pre: The pile is not empty.
// Post: take removes the top Coin and copies
//its data as the return value.
//Remark: All member function test their preconditions
//in an assert statement, so the program must always crash
//with a assert firing in this situation.

void put(Coin& coin);
// Pre: The pile is not full.
// Post: coin is copied to the top of the Pile.

bool empty();
// Pre:
// Post: returns true iff Pile is empty.

bool full();
// Pre:
// Post: returns true iff Pile is full.
// The Pile is full means another Coin cannot be put into it.
// The Pile is full iff used==capacity

void yank();
// Pre:
// Post: The newest coin, closest to the bottom, is removed;
//the coins above it are all moved down one position, and
//the removed coin is copied into the top position.

friend std::ostream& operator<<(std::ostream& outs, const Pile& pile);
// Pre:
// Post: The data in the Coins is output to ostream as follows:
//The Coin data is printed one line for each Coin in the
//order it is stored in the Pile from top to bottom.
private:
    static const std::size_t capacity = 10;
    Coin A[capacity];
    std::size_t used;

    // Representation Invariants of the Pile class:
    // (1) A[] is a partially filled array (as defined in DSO)
    //     with the filled entries occupying positions 0, 1, ..., used-1.
    // (2) 0<=used<=capacity.
    // (3) If the Pile is not empty, A[used-1] holds the Coin that is
    //     at the top of the Pile.

};

#if defined(THEATER_DSO)
#endif

Here are the command letters to be used for the test_Pile test driver user interface.

q To quit the test_Pile program. (It’s wise to implement this one first. It is also wise to
   remember that typing control-c normally “kills” your currently running program.

r A function to re-initialize the pile to the empty state, so the Pile can be reused in a re-
   initialized Game.

n Test the overloaded operator<< function to print the contents of the Pile.

e Test the empty() function. (Of course, print the value returned).

f Test the full() function. (Something should be printed of course.)

t Test the take() function. Print the Coin returned. If the Pile is empty the test program should
   crash because the take() member function should use assert() to check the precondition
   for take().

p Test the put() function. The test driver should prompt for and accept the denomination,
   year and mint mark the Coin to be passed by reference to this function. Again, in some cases
   we should observe an assertion firing.

y Run the yank function. (The user can test its action by giving the n command.)

z Run the sort function. (Ditto.)

The Pile driver should simply call the indicated member functions with parameters supplied by
the user. You (and the graders) will observe how the precondition checking assertions “fire” when
preconditions are violated.
For simplicity, use “cin >> intvar;” to input an integer when an integer is expected: If the program tries to re-read the integer after the user types a non-digit character instead, it will loop infinitely. THAT’S OK for this first project. Future courses and projects might require more robust recovery from user errors.

Here is a sample test_Pile startup message and menu:

unix1% test_Pile
I have initialized an empty Pile of Coins.

The following choices are available:
q  Quit this test program
r  Test reset()
n  Test operator<<( )
e  Test empty()
f  Test full()
t  Test take()
p  Test put()
y  Test yank()
z  Test sort()
Enter choice:

The prompt for the choice in all parts must be exactly:
Enter choice:

(note the colon). The prompt for entering a number must end with a colon. This is critical for getting full credit because the testing system for grading your work will use these prompts to tell when to provide the sample test inputs!

Of course, the testing program must print the return values from those functions that provide return values: empty(), full(), take()

2 Part 2

Below is a sample of what using the teller program should look like. Try out the reference implementation to observe the messages due to user input errors (such as $0.06 coins) and impossible requests.

$ teller
I have initialized 6 empty Piles of Coins.

The following choices are available:
Q  Shut down the bank; ignore the cash
A  Try to accept a coin
R Report drawer contents
Y Yank and put on top a newest coin in each pile
Z Sort each pile
D Disburse a given amount of cents if possible
Enter choice: A
Input the coin (denomination year mint-mark): 5 2003 D
Your coin has been deposited.

....(Menu displays removed to save space on paper)...

Enter choice: A
Input the coin (denomination year mint-mark): 1 1779 a
Your coin has been deposited.

... Enter choice: A
Input the coin (denomination year mint-mark): 1 1890 G
Your coin has been deposited.

... Enter choice: A
Input the coin (denomination year mint-mark): 1 2006 A
Your coin has been deposited.

... Enter choice: A
Input the coin (denomination year mint-mark): 10 1970 D
Your coin has been deposited.

... Enter choice: R

1 cents drawer
1 cents (2006A)
1 cents (1890G)
1 cents (1779a)

5 cents drawer
5 cents (2003D)

10 cents drawer
10 cents (1970D)

25 cents drawer

50 cents drawer
100 cents drawer
...
Enter choice: D
Input how much money you want in cents: 11
Here is your money:
1 cents (2006A)
10 cents (1970D)
...
Enter choice: D
Input how much money you want in cents: 4
Sorry, we do not have the coins to disburse your money.
Enter choice: Q
Ridicule is the best test of truth.
$
3 Project Rules and Guidelines

The use of the following practices will be checked during grading and will contribute to your grade:

1. The project must be done and then submitted as a single directory, possibly with subdirectories if you like. (Each project, lab exercise, etc., must be done under its own directory; so you never worry about which files are for which project.)

2. All core, linked executable files and object files under it MUST be removed before the directory is submitted. (Such files consume disk quota of the course account and so cause problems for all students if some submissions include them. Hence the harsh penalties!)

3. Every implementation file (files with extension .cxx or .template) and non-trivial header file must be accompanied with an RCS database file that records its entire development history. Those RCS database files will be accessed only by RCS commands such as “ci -l” (here “l” is the lower-case letter “ell”). They have names obtained by appending ,v (comma v) to the name of the original file. The database file may exist in an RCS subdirectory of the directory holding the original file, or at the same level as the original file.

The main reasons for requiring version control in this course are:

(a) If (when!!!) you accidently delete, overwrite, irreversibly mess up, or otherwise lose the original file, you can get back old versions of it which you “checked in” (with the ci -l command) before the disaster.

(b) We introduce you to an important software development practice, especially for large or group projects. They use more complex network based revision control systems such as CVS, but RCS is a good place to start.

The quick-start way to use RCS is (after optionally making the RCS subdirectory) start writing filename with the text editor, and then, each time you save the file, give the command “ci -l filename”. That command will prompt you to write a comment about the version you are “checking-in”. You can then review the history of the versions with the command “rlog filename”.

If (when!) disaster strikes, read the man pages and/or consult a TA, knowledgable classmate, or the professor for instructions on using certain other RCS commands to retrieve an older version.

There is a handy emacs command “C-x v v” that automates somewhat more sophisticated RCS usage. Type this command while in the buffer editing the file in order to save the file; and then type it again in order to begin modifying the file again. When you command “C-x

---

1OK, here’s how to do it. (1) ci filename to check in the bad version, to unlock it and to keep it just in case you will need it. (2) co -r1. n filename to retrieve an older version, say 1.n, read-only. (3) Is it the one you can use? If not, repeat step (2). (4) rcs -l filename to “lock it”. (5) ci -l filename to check the “reverted version” back in as the latest version. This makes filename writable again, and you can continue from before the disaster! For further info, begin with man rcsintro.
v v” after editing, **emacs** will open another buffer for you to write a version comment: After typing a comment about this version, press Control-c twice (C-c C-c).

4. Header (class declaration) files, function implementation files, and template files (when we get to them) for each C++ class you write MUST all be separate, and be separate from files that implement test drivers. Many examples of separating parts of a program into multiple files, each with a small, well-understood purpose, are given in the DSO textbook. (The Lab 1 exercise provides a simple first example.)

5. A “build shell script” named `build.sh` must be provided at the top level of the submitted project directory. This build script must contain the commands that build the project software so we can test it. (The Lab 1 exercise covers how to do this.)

6. **“ZERO GRADE IF IT DOESN’T COMPILE”**

When we grade a submission, we will recreate the directory you submitted. First we will delete all object and executable files (and subtract a grade penalty if there were any). Next, we will run the `build.sh` script in its directory. If the build script fails to make your submission compile and link to specified executable files we can run, the project (or part) will get ZERO points AUTOMATICALLY.

### 4 Elementary Software Development Practices for CSI310

Follow the advice of section (1.3)! It is wise to not “change suspicious code on the hope that the change might work better.” But here’s one addition: Sometimes, usually at the beginning of a project, the code you wrote might be so confusing to yourself that is impossible to tell what is causing failure, even when program execution is examined with a debugger. In that case, it might be a good idea to *throw out* your first or so attempts and *start over again*! It will not be a waste of time because the first or so attempt at a concrete design makes your mind really understand the problem.

Some students have been taught to write each programming assignment all at once, trying to get it correct before beginning to typing it. Usually it is not correct, so hours of syntax error correcting and then debugging ensue. This practice is immature and practically useless, except for extremely small problems, and for assignments meant to teach the simplest things (and for CS course exams).

After some careful specification, design and perhaps analysis, “interesting” programs are best written (i.e., implemented) **incrementally**: A compilable early version that doesn’t do much is created very quickly. Then, parts are added little by little, in an order designed to **enable testing and debugging of each new part as soon as it is added**. Often, test driver `main()` functions are written for the sole purpose of testing the parts (i.e., classes, when you use C++ or Java) you are going to write next. After a part or class is tested, debugged, and tested again, we can begin implementing a new class that **USES** it.

Thus, part 1 of this project begins by using a declaration, and writing a test driver and then implementations of class `Coin` and class `Pile`.

---

11
Part 2 is to develop a main function and functions it calls (named teller) which interacts with the user, uses 6 Piles to manage the storage of each denomination of Coin separately, and implement the teller simulation. So, a software engineer would say that the teller module uses the Coin and Pile classes.

5 Sample Class Design and Implementation Plan Outline

Here we give concrete guidance to this approach. It is explained in terms of the simple class “throttle” that is the subject of Chapter 2 and Lab 1.

Tip: Check the check boxes as you proceed through the first part of the first project, to get the idea of how to work this way. In some professional programming groups and for big enough programs, management will demand a report at each step!

1. □ Study the assignment (specification) and sketch (perhaps on paper):

   (a) □ The name of the class (e.g., throttle).

   (b) □ Names of public member functions, notes clarifying what each does, what arguments and return values it should have. Perhaps sketch diagrams.

   (c) □ REVIEW your sketchwork and the assignment to correct, improve and clarify the sketch. Ask yourself if the member names suggest their purpose as clearly as possible. If not, change them. For example, the throttle’s shift function is pretty well-chosen; to name it set would be misleading (why?)

2. □ Type the header file (e.g., throttle.h) to contain formal documentation and the class definition.

   (a) □ Document each public member function with its name, prototype and (IMPORTANT!) comments in the form of pre and post-conditions to document what each public member function is supposed to do.

   (b) □ Write the class definition’s public member function declarations only (no private members yet!)

3. □ Create a skeleton implementation file for the class (e.g. throttle.cxx) in which the bodies of member functions are stubs.

   (a) □ A stub is trivial function body that might be empty, or perhaps more usefully, prints a message that it is called. It might also print its arguments.

   (b) □ Make it return some reasonable constant that’s trivial to express, if the return type is not void.

   (c) □ The skeleton implementation file MUST have directive to “include” the class’s header file. For example:

   ```
   #include "throttle.h"
   ```
4. Write a test driver (like demo1.cxx or demo2.cxx).

(a) The test driver uses the class you are developing, so it MUST have a #include preprocessor directive to “include” the class’s header file. That directive is identical to the one in the header file! (Just copy it—copy and paste makes that easy and error-free).

(b) Try to compile the test driver.

   Correct it until it compiles.

   This step might show you something was wrong with the declarations in the header file. If so, correct the header file; but, if you do, (IMPORTANT) try to compile the stub implementation file again! Make corrections there too if necessary.

(c) Try to link the test driver and class implementation object files. E.g. with the command “g++ -o demo2 demo2.o throttle.o”

   i. Edit and recompile and relink all until it succeeds.

   ii. See if the test driver runs as you would expect with the stub implementations of the tested class. In other words, begin by testing the tester! Debug, recompile, relink till it’s right.

(d) NOW (finally!) begin coding the private data member declarations according to your data structure design choices, and implement the member functions.

   i. Code the data members first.

   ii. Code one or more constructors: Recompile, relink, and debug until you verify the constructors work.

   iii. Replace the stubs (one or a few at a time) with real implementations. Begin with functions that display or use class data without modifying it. Recompile, relink and use the test driver to test what you have implemented as soon as possible.

   iv. Choose new stubs to replace in an order that enables you to use previously written and tested functions to test the new ones. Overloaded operator>>( ) functions or other input functions might be good choices so you can use them to input more cases of class data.