CSCI 104 - Spring 2016 Data Structures and Object Oriented Design

Data Structures and Object Oriented Design

- Due: Friday, March 25th, 11:59pm
- Directory name for this homework (case sensitive): `hw5`
  - This directory needs its own `README.md` file
  - You should provide/commit a Qt `.pro` file. We will then just run `qmake` and `make` to compile your `search` program.
    - In your `.pro` file add the following lines:

```
CONFIG += debug               # enables debugging (like the -g flag)
TARGET = debugger               # sets output executable name to debugger
QMAKE_CXXFLAGS += -std=c++11  # enable C++11 libraries
SOURCES -= msort_test.cpp   # Don't compile certain test files
```

- Start early!

Overview

For this (second) part of the project, we will focus on using Qt to build a graphical debugger on top of your Interpreter. In creating a Qt interface, you will practice various more complex relationships between objects, along with inheritance. We will describe it in more detail below. **The first problem is a standalone problem on class hierarchy, inheritance, and composition. It doesn't affect any of remaining problems.**

Step 1 (Class Structures and Inheritance, 15%)

This problem has nothing to do with your project, but it will separately test your understanding of class structures and inheritance.

Imagine that you are writing (yet another) car racing game. In it, the player can customize her race car in a number of ways:

- there are (currently - future versions might add more) three different type of chassis available: the Truck, the Racer, and the Buggy. They drive differently.
- the player can customize the color of her car.

- there are (again: currently) three different engines available: the Nitrox, the Warp, and the PowerBlast. They have different properties.
- the player can accumulate arbitrarily many "Items". These currently are
  - Car Stereo
  - Spoiler
  - Bike Rack
  - Flower Vase
  - The following Weapons: Harpoon, Machine Gun, Grenade Launcher, Oil Dispenser
- While a player can accumulate arbitrarily many weapons, at most one can be in use at a given time.

The players also have various characteristics, like strength, experience, and control. The game supports arbitrary numbers of players playing at any given time.

Diagram the classes involved in the game so far. (We will ignore such things as racetracks and other parts of the game.) Indicate which classes are abstract, and which aren't. Show which classes inherit from each other, publicly or privately. Also show which classes have a "has-a" relationship, possibly using sets or maps or lists where appropriate.

You should identify the key virtual functions (and whether they are pure virtual functions). Also, identify the data members of a class insofar as they indicate "has-a" relationships.

You can choose to draw this by hand and scan it (and submit as a JPG or PDF), or to use some graphics software (and produce a JPG or PDF), or to use UML (if you know it), or draw it using ASCII art (this may be a lot of work). Provide an explanation with your choices, i.e., tell us why you chose to have certain classes inherit from each other (or not inherit). (Notice that you don't need to do any actual programming for this problem.)

Of course, there isn't just one solution, but some solutions are better than others. Just do your best to write a class hierarchy, define member functions, and identify key composition and data structure members to support the description above. Good explanations may help you convince us that your proposed approach is actually good.

Step 2 (Comparator Functor, 0%)

The following is background info that will help you understand how to do the next step.

If you saw the following:

```
int x = f();
```

You'd think `f` is a function. But with the magic of operator overloading, we can make `f` an object and `f()`

a member function call to `operator()` of the instance, `f` as shown in the following code:

```
  struct RandObjGen {
    int operator() { return rand(); }
  };

  RandObjGen f;
  int r = f(); // translates to f.operator() which returns a random number by
calling rand()
```

An object that overloads the `operator()` is called a **functor** and they are widely used in C++ STL to provide a kind of polymorphism.

We will use functors to make a Merge Sort algorithm be able to use different sorting criteria (e.g., if we are sorting strings, we could sort either lexicographically/alphabetically or by length of string). To do so, we supply a functor object that implements the different comparison approaches.

```
  struct AlphaStrComp {
    bool operator()(const string& lhs, const string& rhs)
    { // Uses string's built in operator<
      // to do lexicographic (alphabetical) comparison
      return lhs < rhs;
    }
  };

  struct LengthStrComp {
    bool operator()(const string& lhs, const string& rhs)
    { // Uses string's built in operator<
      // to do lexicographic (alphabetical) comparison
      return lhs.size() < rhs.size();
    }
  };

  string s1 = "Blue";
  string s2 = "Red";
  AlphaStrComp comp1;
  LengthStrComp comp2;

  cout << "Blue compared to Red using AlphaStrComp yields " << comp1(s1, s2) <<
endl;
  // notice comp1(s1,s2) is calling comp1.operator() (s1, s2);
```

```
  cout << "Blue compared to Red using LenStrComp yields " << comp2(s1, s2) << endl;
  // notice comp2(s1,s2) is calling comp2.operator() (s1, s2);
```

This would yield the output

```
1  // Because "Blue" is alphabetically less than "Red"
0  // Because the length of "Blue" is 4 which is NOT less than the length of "Red"
(3)
```

We can now make a templated function (not class, just a templated function) that lets the user pass in which kind of comparator object they would like:

```
template <class Comparator>
void DoStringCompare(const string& s1, const string& s2, Comparator comp)
{
  cout << comp(s1, s2) << endl;  // calls comp.operator()(s1,s2);
}

  string s1 = "Blue";
  string s2 = "Red";
  AlphaStrComp comp1;
  LengthStrComp comp2;

  // Uses alphabetic comparison
  DoStringCompare(s1,s2,comp1);
  // Use string length comparison
  DoStringCompare(s1,s2,comp2);
```

In this way, you could define a new type of comparison in the future, make a functor struct for it, and pass it in to the `DoStringCompare` function and the `DoStringCompare` function never needs to change.

These comparator objects are used by the C++ STL `map` and `set` class to compare keys to ensure no duplicates are entered.

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

You could pass your own type of Comparator object to the class, but it defaults to C++'s standard less-than

functor `less<T>` which is simply defined as:

```
template <class T>
struct less
{
  bool operator() (const T& x, const T& y) const {return x<y;}
};
```

For more reading on functors, search the web or try this link

Step 3 (Implement Merge Sort, 25%)

Write your own template implementation of the Merge Sort algorithm that works with any class `T`. Put your implementation in a file `msort.h` (that is, don't make a `msort.cpp` file). Your `mergeSort()` function should take some kind of list (you probably want to choose `vector<>`, but feel free to choose something else). Your `mergeSort()` function should also take a comparator object (i.e., functor) that has an `operator()` defined for it.

```
  template <class T, class Comparator>
  void mergeSort (vector<T>& myArray, Comparator comp);
```

This allows you to change the sorting criterion by passing in a different Comparator object.

You should test your code by writing a simple program that defines 1 or 2 functors and then initializes a vector with data and finally calls your `mergeSort()` function. You should be able to produce different orderings based on your functors. You should submit this test code.

You are free to define a recursive helper function so that your main `mergeSort()` just kickstarts things by calling the helper function.

Step 4 (Write a graphical debugger using Qt, 60%)

You will build on your interpreter from part 1 of the project to also provide a graphical debugger. This will require some (not too extensive) changes to the structure of your Interpreter. You should provide a nice GUI to help the user debug their Facile program. Feel free to get creative with the design. You WILL be deducted points if your windows are arranged in an unappealing layout. The minimum specifications are as follows.

Load Window

When the program starts, it should present you with a window that asks the user to type in a filename into a textbox. You should provide a quit button which, when pressed, causes the program to gracefully

terminate. You should also provide a button which, when pressed, loads the file of the name provided in the textbox. If the user instead presses enter in the textbox, this should also load the appropriate file.

If there is an error in opening the file, you should show an Error Window with an appropriate error message on it for the user, as well as a button which, when pressed, hides the Error Window. If you open the file successfully, you should hide the Load Window and show the Debugger Window.

Debugger Window

You must provide (at least) the following:

- A scrolling box which allows the user to inspect the entire Facile program by scrolling up or down. The user should be able to select a specific line of code in this box.
- Some form of visual cue in the scrolling box to let the user know which line of code the interpreter has halted on.
- A breakpoint button which, when pressed, adds a breakpoint to the selected line of code. There should be some kind of visual cue to let the user know which lines currently have breakpoints.
- A continue button. This causes the program to continue to interpret the program until it reaches a breakpoint, at which point it should wait for further user input.
- A step button. When the step button is pushed, the interpreter should advance one line of code, and then wait for further user input.
- A next button. This operates exactly like a step, except that if it is on a GOSUB line, then it should interpret until it returns from this GOSUB call, or until it reaches a breakpoint (whichever happens first).
- An inspect button. When pressed, this should show the Values Window.
- A quit button. When pressed, the program gracefully terminates.
- If an error message (such as divide by 0) occurs, you should show an Error Window with the appropriate error message. This window should have a button which, when pressed, hides the Error Window. You do not need to display more than one Error Window (if the user never hides a previous Error Window, you can reuse it but replace the message).

Note that you will need to change your interpreter so that instead of automatically advancing to the next line to interpret, it has the possibility of pausing and waiting for user input.

Additionally, when the Facile program terminates, your Debugger should still be running. You should reset everything to the starting point of the Facile program (set the current line to the first line, clear your map, etc) so that the user can run through the program again if they choose to.

Values Window

This window should display all of the current values of the variables inside your map, inside a scrolling box. There should be 2 buttons, one which hides the values window, and one which updates the values window. There should be four options (which the user can choose via a dropdown box or radio buttons or

equivalent) where the options are to sort the variables by name (ascending order), sort by name (descending order), sort by value (increasing order), or sort by value (decreasing order). If a new option is selected, the variables should not be re-arranged until after the Update Button has been pushed. When the window appears for the first time, the variables should be sorted by name.

If the user leaves the window open while continuing to interpret the program, you do not need to update the values window. You should instead update the values window when the Update Button is pushed (or when the Values Window is opened).

You need to use your MergeSort function to implement the sorting. You will need to read the entries from your map into a data structure of your choice (probably a vector of pairs), and you will need to pass in an appropriate comparator for the specific sorting criteria that is being requested.

QT Notes

A single class can act as multiple windows by simply having several "Widgets" as either data members (or if your class inherits from QWidget or QMainWindow your object itself can be a window). Any QWidget that is not a "child" of another widget (i.e., added to a layout that is part of another widget) can act as a top-level window. Simply call `show()` and `hide()` to make the window appear and disappear.

Another important tip is to NOT create a window each time you want to open it. Essentially, create it once at startup and just `show()` it when you need it, `hide()` it when you want it to "close", and simply call `show()` again when you want it to reappear. Before you call `show()` just populate the windows controls with the updated data you want to display. Don't reallocate and delete a Widget/window/control multiple time. The bottom line: **It is best to allocate widgets/controls only once at startup and never again**.

Here is an example of the above idea:

multiwin.h

```
#ifndef MULTIWIN_H
#define MULTIWIN_H
#include <QWidget>
#include <QPushButton>
#include <QLabel>

class Multiwin : public QWidget
{
  Q_OBJECT
public:
  Multiwin();
public slots:
```

```
  void mainButtonClicked();
  void otherButtonClicked();
private:
  QPushButton* mainButton;
  QWidget* otherWin;
  QPushButton* otherButton;
};
#endif
```

multiwin.cpp

```
#include <QVBoxLayout>
#include "multiwin.h"

Multiwin::Multiwin() : QWidget(NULL)
{
  QVBoxLayout* mainLayout = new QVBoxLayout;
  mainButton = new QPushButton("&Open OtherWin");
  mainLayout->addWidget(mainButton);
  setLayout(mainLayout);

  QVBoxLayout* otherLayout = new QVBoxLayout;
  otherWin = new QWidget;
  otherButton = new QPushButton("&Close");
  otherLayout->addWidget(otherButton);
  otherWin->setLayout(otherLayout);
  QObject::connect(mainButton, SIGNAL(clicked()), this, SLOT(mainButtonClicked()
));
  QObject::connect(otherButton, SIGNAL(clicked()), this, SLOT(otherButtonClicked
()));
}

void Multiwin::mainButtonClicked()
{
  otherWin->show();
}
void Multiwin::otherButtonClicked()
{
  otherWin->hide();
}
```

Layouts and widgets that you don't need to access after creation do **not** need to have to be assigned to a data member in the class. They can just be created using a temporary pointer and added to some other layout/widget.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your `hw5` directory to your `hw_usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2016/hw_usc-username.git`
5. Go into your hw5 folder `$ cd hw_username/hw5`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.