CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

- Due: Friday, October 23, 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 = search               # sets output executable name to search
QMAKE_CXXFLAGS += -std=c++11  # enable C++11 libraries
SOURCES -= msort_test.cpp # Don't compile certain test files as part of
the search application
```

- Warning: Start early!

Overview

For this (second) part of the project, we will focus on using Qt to put a graphical user interface on your seach engine. 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 a simulation of a restaurant to help make decisions about number of employees, ensure appropriate crowding levels both in the restaurant and kitchen, etc.

- The restaurant employs chefs, waiters, and hosts. The restaurant also has an owner.
- Employee pay is calculated using a different algorithm based on the type of employee. Waiters also receive tips. The owner doesn't get paid hourly ... she owns the business.
- Parties of customers are categorized as walk-in, reserved, and celebrity parties
  - A party has a size (number of people).
  - A party will be matched to a table (we'll assume 1 party per table).
  - A party has a time that they will take to order and also a time they will take to eat/pay/exit.

  - All parties tip, and the algorithm for calculating the tip is based on their category (celebrities generally tip 25% + a constant amount while walk-ins usually tip 15% for other parties of 2-3 people, 18% for parties of 4 or more, etc.).

- Each waiter has a list of tables he/she is assigned to wait on.
- Each chef has a different skill level allowing them to cook the order faster (higher skill) or slower (lower skill).
- Each chef has a queue of orders they are assigned to cook.
- Waiters take orders from a particular table, enter the kitchen, give the order to a chef (don't worry about how they choose the chef), who indicates how long it will take and then the waiter returns to the dining area. The waiter will return to the kitchen to get the order after that time as elapsed.
- The restaurant should support the following operations:
  - Return a list of all the employees.
  - Compute the gross amount of money/pay employees earned by iterating through the list of all employees
  - At any particular time, indicate how many customers a waiter is serving.

Note: You don't have to worry about how time would "increase" or "pass", nor do you have to worry about the overall coordination of who calls the higher level functions. Just consider the objects and member functions necessary to support appropriate invocation of the described functionality.

Diagram the classes (show as a box) involved from the description given. Indicate which classes are abstract, and which aren't. Show which classes inherit from each other with an array and label it as `public` or `private`. Also show which classes have a "has-a" relationship to another class and indicate what container class (if any) would be used to store them (e.g., lists, sets or maps 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.

For this problem, 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 keep 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 (choose either `vector<>`, `list<>`, or `deque<>` based on the needs of your project). 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 (list<T> or vector<T> or deque<T> myArray, Comparator comp);
  /* you may choose the type of list/vector/deque that fits your design best */
```

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 list/vector/deque with data and finally calls your `mergeSort()` function. You should be able to produce different orderings based on your functors.

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

Step 4 (Put a Qt frontend on it, 60%)

Replace your `cin`, `cout` based user interface by a Qt interface. When the program starts (the command line arguments are still the same), it should present you with a window that contains widgets pertaining to the search operation as well as a blank listbox that will be populated with the filenames of hits that the search returns. Specifically your interface should have:

- A textbox to enter search terms separated by spaces
- Radio or other toggle buttons to select single, AND, or OR search
- A pushbutton to initiate the search (i.e. a "Search" button). Pressing return in the search term textbook should also initiate the search.
- The results of the search should populate a scrolling listbox with the filenames and numbers of incoming and outgoing links of all matching webpages:
  - The default should be that they are **sorted by filename** (in alphabetic order with a's at the top of the list and z's at the bottom). However, there should be buttons (or a radiobutton) to let the user change the sorting criterion between sorting by filename, by number of incoming links, or by number of outgoing links (from the page with the smallest number of links at the top to largest at the bottom)
  - For that reason, you must use your mergesort implementation with appropriate functors to sort the results and display them in correct order
- The user should be able to select a filename in the list of results and display that webpage in a separate window, either by double-click, select & pushing a separate button, or some other method that is "reasonable".
  - When the new window opens, it should display the name of the file on top
  - The separate webpage display window should display the selected page's contents (using the same display rules as the previous assignment).
  - In addition, the separate webpage window should have two lists showing both the incoming and outgoing links of that page. Those file names should be sorted alphabetically, by number of incoming links, or number of outgoing links (as chosen by the user); for that reason, you should again use your own `mergeSort()` implementation. When a user chooses a particular page linking to or from the current one (again you can choose to allow the user a double-click, pushbutton, etc.), it should update the window to display that file's contents, outgoing links, and incoming links

(i.e., replace the contents of that window with the information from the newly selected file).
    - If you want, you can get rid of the outgoing links window, and instead display clickable hyperlinks in the text that is shown in the main window to the user. The reason we are not making this the default is that it is quite a bit more difficult to write. Consider yourself warned!
    - There should be a "close" button to close the webpage display window (or hide it).
- If the user performs a new search query back in the main window, it should repopulate the listbox of hits in the main window. However, it need not clear the contents of the second window (if it is already open) until the user selects a new file.
- There must be a way to exit the program with a "quit" button on the main page.

Here are two datasets you can use to test your code: one with 10 pages and one with 100 pages.

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.