COM S 213 – Spring 2003

ASSIGNMENT #5: **Inheritance in the Menu Class**

DATE GIVEN: **2/28/03**

DATE DUE: **3/7/03**

PURPOSE:
To apply what we've learned about inheritance to our Menu class.

ASSIGNMENT:
For this assignment we are going to re-implement our Menu class using inheritance. Up until now, we've viewed the items in our Menu as either hard coded or string data. Now, we will treat each menu item as its own data type. And, while we're at it, we'll tackle another problem we haven't dealt with yet… Menus can be hierarchical—that is—a menu item could be *another menu*! But, if we are going to store all menu items as one data type (presumably in a dynamic array) how can we allow for two different data types in the array (Menus and MenuItems).

Here's where inheritance helps us out. We can create an abstract class named MenuObject from which Menu and MenuItem can be derived from. Then, in any Menu class, the dynamic array of menu items is actually an array of MenuObject *pointers*. We need to use pointers for two reasons. (1) We'll be using virtual functions and overriding, so we can't use static instances of MenuObject and achieve our desired behavior, and (2) we will put some pure virtual functions in the base class which would make it impossible to have standalone instances of it in the array!

Let's start by defining the base class functionality. What features of Menu and MenuItem are common across classes? Well, both will have some sort of string name associated with the object, both will need to be displayed to the user (when their "parent" menu is being displayed), and both will need to do something when "selected". With this in mind, a Menu should be displayed slightly differently then a MenuItem so that the user has some visual indicator that the menu choice is really another Menu. We'll display MenuItems that are really nested menus by putting the name in square brackets. So, this leads us to the following structure for the MenuObject:

MEMBER TYPE	MEMBER NAME	DESCRIPTION
Function	`display`	When called, the name of this object will be displayed. If the object is a MenuItem, just the name is displayed. If the object is another Menu, the name will be displayed in square brackets. The actual displaying is taken care of in the overridden version of this function in the derived classes. This function should be *pure virtual*.
Function	`select`	When called this function should do something to indicate that this MenuObject has been selected. In the case of MenuItems, the message "You chose…" followed by the name of the menu item is displayed. In the case of another Menu, The nested menu is "run". So, Menu::Select() in this assignment is very similar to Menu::Run() in our earlier assignments.
Variable	`name`	This string variable simply holds the name of the MenuObject. This will be used from Display() in the derived classes, so it should either be protected or have a public getter function.

The MenuObject class should also have a constructor which allows the name member variable to be set when created.

Let's look at the MenuItem class next. It is fairly simple. It should contain the following:

MEMBER TYPE	MEMBER NAME	DESCRIPTION
Constructor	`MenuItem`	Takes an initial string to name the MenuItem. This gets passed along to MenuObject's constructor.
Function	`display`	Simply displays the menu item's name. All we're doing is printing out the name, no newlines.
Function	`select`	Simply echoes a message out to the console that says "You chose…" followed by the MenuItem's name.

Now, let's look at the Menu class. It is a little more complicated, as you'd imagine:

MEMBER TYPE	MEMBER NAME	DESCRIPTION
Constructor	`Menu`	Takes the name and size of the menu. The name is passed along to MenuObject's constructor and the dynamic array of MenuObject pointers is allocated.
Variable	`maxsize`	The maximum number of MenuObject pointers allowed in our array (the total number of entries allowed in this

		menu)
Variable	`cursize`	The total number of MenuObjects added to this menu (the total number of entries in the menu so far)
Function	`display`	Displays the menu name (**not** this menu's entries). Wrap the name in square brackets so that the user can see that this entry represents a menu and not just another MenuItem.
Function	`select`	When called, this menu should be "Run". All MenuObjects in this menu should be displayed. An additional entry should be displayed which lets the user "Exit" from this menu. All menu objects are displayed by looping for each item in this menu's array of MenuObject pointers and calling MenuObject::Display for each. Prefix each call to MenuObject::Display with a number that can be used to identify the object (the number the user chooses when selecting this entry). When the user selects an item, call MenuObject::Select() for that entry. If the user chooses the Exit option, terminate your loop. This function is roughly equivalent to Menu::Run() from assignment #4.
Function	`addItem`	Takes a MenuObject pointer as an argument and adds that MenuObject to the this Menu if there is sufficient space left.

OK, this does not give us a completely functional Menu system yet, but gets us very close. With what is described above we can have a menu system displayed that allows us to navigate down all nested sub menus and let's us come back up the chain one at a time. Future assignment will have us refine this system even further.

BUT, with what we've described so far, we could have a main() function which builds and kicks off a mock Bear Access style menu:

```
void main()
{
  // Build our menus
  Menu *topMenu = new Menu("Top Level",2);
  Menu *baMenu = new Menu("Bear Access",4);
  Menu *dirSvcsMenu = new Menu("Directory Services",2);
  Menu *salsaMenu = new Menu("Project SALSA Tools",3);

  topMenu->addItem(baMenu);
  topMenu->addItem(salsaMenu);

  baMenu->addItem(new MenuItem("Eudora"));
```

```
  baMenu->addItem(dirSvcsMenu);
  baMenu->addItem(new MenuItem("CUInfo"));
  baMenu->addItem(new MenuItem("Colts II"));

  dirSvcsMenu->addItem(new MenuItem("Who I Am"));
  dirSvcsMenu->addItem(new MenuItem("Directory Search"));

  salsaMenu->addItem(new MenuItem("Server Manager"));
  salsaMenu->addItem(new MenuItem("Runway"));
  salsaMenu->addItem(new MenuItem("Agent Harry"));

  // Start the menu running!
  topMenu->select();

  delete topMenu;
  delete baMenu;
  delete dirSvcsMenu;
  delete salsaMenu;
}
```

You assignment is to implement the classes as defined in this assignment so that this main function can be used to test your work.

USEFUL HINT:

In doing this assignment you will probably want to include MenuObject.h from Menu.h and MenuItem.h. You will also need to include Menu.h and MenuItem.h from main.cpp. This creates a situation where MenuObject.h ends up being included twice in the same .cpp file (main.cpp).

You can get around this by "wrapping" the MenuObject.h file with compiler directives, like this:

```
//
// MenuObject.h
//

#ifndef __MENU_OBJECT_H
#define __MENU_OBJECT_H


// put the contents of MenuObject.h here
```

#endif

What this does is tell the compiler to include MenuObject.h only if it hasn't seen it before when compiling a particular file. If you'd like further explanation of how this works, please contact me.