EECS 211 Program 3
Due: Sunday, April 29, 2012

In this first program in our project we will begin to implement two of the classes we need – **SYMBOL** and **symbolList**.

Background:

Most applications have thousands or millions of lines of code, much more than would fit in a single file. So projects are built from many moderate-sized files. Moreover, it is standard to organize the software into related pieces that fit together in one or a small number of files. The most common such strategy is to put the declarations and code for a class or a small group of related classes into a pair of files – a header file (.h file) containing the declarations and a code file (.cpp file) containing the implementation of those classes. Another common practice is to use a pair of files, .h and .cpp, for utility functions – functions used by the main algorithms but not directly part of the main algorithms. Finally, larger projects use common definitions of symbols and parameters, such as length of array or error conditions, so that all files in the project will be coordinated. In our project we will follow these standards. Even this first assignment of the project will have a total of eight files.

When there are many header (.h) files, it is easy to have multiple or even circular inclusions – one file includes a.h, a.h, includes b.h, …, and eventually some file includes a.h again. When this happens, the compiler will see multiple definitions of symbols and classes, which is an error. The standard C mechanism to prevent this from causing errors is to use two additional # keywords, #ifndef (if not defined) and #endif (end of text started by a #ifndef). These two are used with a symbol (usually the name of the header file in upper case and underscore in place of the dot) as follows. Suppose we have a header file called mystuff.h. The format of that file would be:

```
#ifndef MYSTUFF_H
#define MYSTUFF_H

…  all the regular declarations, etc.

#endif
```

Suppose some cpp file, say a.cpp, would include mystuff.h twice. During the compilation of a.cpp, at the first inclusion of mystuff.h the symbol MYSTUFF_H would not have been seen, the #ifndef condition would be true, and all the lines down to the #endif would be processed. In particular, the #define MYSTUFF_H line would be processed, adding the symbol MYSTUFF_H to the compilers symbol table. At the point where mystuff.h would be included a second time, the #ifndef condition would fail because MYSTUFF_H was added to the compiler's symbol table during the first inclusion. When a #ifndef condition fails, all the lines to the matching #endif are

skipped. Thus, with this mechanism programmers are freed from having to struggle to remember which header files are included in which other header or cpp files.

We will also introduce the notion of validity checking for data and the use of error codes as return values for functions. In robust systems and applications it is necessary for upper level functions to know whether lower level functions have succeeded or not. In case a lower level function fails to perform its function, the upper level function must take some corrective action. A simple example familiar to most students is an ATM. If the cash dispenser function fails to dispense the cash, the ATM should take some corrective action, like display a message to the user and send a message to the maintenance department. You are also already using this because your own main function in your projects returns 0 when it's done. By default, 0 is used to indicate success, and non-zero return values indicate errors. Most systems define a large number of error codes so that top level portions of the system can recover properly depending on what the error was.

Assignment:

(1) Your project for the assignment will consist of eight files – definitions.h, main.cpp, symbols.h, symbols.cpp, symbol_list.h, symbol_list.cpp, system_utilities.h and system_utilities.cpp.

(2) In the file definitions.h define the following symbols to have the corresponding values:

SYM_LIST_FULL	21
DUPLICATE_SYMBOL	22
SYMBOL_NOT_FOUND	23

(3) Declare and implement the following class. The declaration should be made in the file symbols.h, and the implementation done in the file symbols.cpp.

Class **SYMBOL**:

- data members:
  - **char** pointer – points to a set of bytes that represent the name of the symbol.
- function members:
  - A constructor which takes as argument a pointer to **char**. The argument is the name of the symbol. The constructor mallocs (see **Requirements and Specifications** section for further discussion of malloc) new space for the name and then copies the name from the argument pointer to the newly allocated space.
  - A destructor. This function frees the memory allocated for the name when the object was created.

- A function called **print**. This function prints the name of this symbol.
- A function called **isThisMyName** which takes a pointer to **char** as argument. The argument points to a string. This function returns 1 if the argument is the same as the symbol name and 0 otherwise.
- A function called **copyMyName** which takes a pointer to a pointer to **char** (i.e., **char** **n) as argument. This function mallocs enough space to hold the string representing the name and assigns that address to the argument. It then copies the name to the newly allocated space.

Although there are five functions, none of them should be very long.

(4) Declare and implement the following class. The declaration should be made in the file symbol_list.h, and the implementation done in the file symbol_list.cpp.

Class **symbolList**:

- data members:
  - array of pointers to SYMBOL – holds a list of SYMBOL objects. The length of the array is LISTLEN (see below).
  - **int** – used to hold the number of symbols currently in the list.
  - **int** – used to indicate whether or not the list can have duplicate symbols, i.e., two different elements that have the same name.
- function members:
  - Constructor. The constructor should have one parameter – an **int** telling whether or not the list can contain duplicates. This argument will be one of ALLOWDUPLICATES or DONTALLOWDUPLICATES (see below). The constructor should copy the argument to the appropriate data member and set the number of elements in the list to 0;
  - Destructor. The destructor should destroy any SYMBOL objects in the list.
  - A function called **print**. This function has no arguments. If the list has no elements, print an informative message. Otherwise, print the heading "Printing SYMBOL list:" on one line and then print each symbol in the list on a new line, indenting the symbols five spaces.
  - A function called **addSymbol**. This function takes a pointer to **SYMBOL** as argument. If the list is full, this function returns the defined constant SYM_LIST_FULL. If the list is not full, then if this list does not allow duplicates, check to see if a SYMBOL object with the same name as the argument already occurs. If so, return the defined constant DUPLICATE_SYMBOL. Otherwise, add the symbol to the end of the list and return 0.
  - A function called **getSymbol**. This function takes two arguments – a pointer to character indicating the name of the symbol to be retrieved and a pointer to pointer to **SYMBOL** (i.e., **SYMBOL** **s) to receive a copy of the symbol. If no symbol with the indicated name occurs in the list, return the defined constant SYMBOL_NOT_FOUND. Otherwise, malloc space for a symbol

into the second argument pointer and copy the first occurrence of the symbol to the newly allocate space.

- A function called **removeSymbol**. This function takes one argument, a pointer to **char** indicating the name of the symbol to remove from the list. If no symbol with the indicated name occurs in the list, return the defined constant SYMBOL_NOT_FOUND. Otherwise, remove the first occurrence, destroy the SYMBOL object, move all the other symbols down in the array, and return 0.

The header file symbol_list.h should use #define to give the value 10 to LISTLEN, 1 to ALLOWDUPLICATES, and 2 to DONTALLOWDUPLICATES.

(5) In the file system_utilities.h declare a prototype for a function called printError, and then implement this function in the file system-utilities.cpp. The details are as follows:

- This function has no return value.
- This function takes a single argument of type **int**. The argument will have one of the values indicated in item (2) in this section, although more will be added in later assignments.
- This function prints an appropriate message depending on the value of the argument on a separate line on the screen.
- The function should use the **switch** mechanism to branch to the appropriate output statement. There should be a **default** case that prints "***ERROR***" on a separate line of the screen.

(6) Use the main function that is posted on Blackboard. This main function will be used only for this assignment.

Requirements and Specifications:

(1) Symbols that are added to a symbol list will be created in the main function with the C++ **new** operator. Therefore, in the destructor for the **symbolList** class these objects should be destroyed by using the **delete** operator. On the other hand, the space allocated for symbol names will be obtained from the system using the **malloc** function. Therefore, the destructor for the **SYMBOL** class should use the **free** function to give this space back to the system. The prototypes for malloc and free are

(void *) malloc(int *number_of_bytes*);
and **void free((void *) *address_to_be_freed*);**

They are declared in the built-in C++ header stdlib.h. (Note, most compilers require the .h in the #include statement for stdlib!) The malloc function takes an integer as parameter and returns an un-typed pointer (i.e., type **void***) to a block of bytes. Note,

the name data member of the **SYMBOL** class is of type (char *), so the assignment statement that allocates the memory has to cast the return type of malloc to be a character pointer. So, the constructor might include a statement like:

```
name = (char *) malloc( length +1);
```

The cast "(char *)" tells C++ that you know you are requesting it to treat the un-typed pointer returned by malloc as if it were a character pointer. The free function does not return a value, so all you need to do is call it with the pointer to the memory block you want to give back to the system. For example:

```
free( (void *)name );
```

where name is of type **char***. Note the reverse cast – from character pointer to un-typed pointer.

Comments, suggestions, and hints:

(1) Just a reminder that it is quite common to write a "throw away" test driver. For testing an individual class in isolation from the other parts of the project, as is the case in this assignment, the throw-away approach can be used effectively. For testing groups of interacting classes or for simulating real-time or event-driven systems, the command-line approach that we will develop in programs 4 and 5 can be very effective.

(2) The main function posted on Blackboard tests all the functionalities of your classes. As usual, I am suggesting that you build your classes in stages and only implement a few of the functions at a time. Therefore, you should comment out the sections of our test main function that call functions which you have not yet added to the classes or implemented. After you have debugged one set of functions and begin developing another set, you can un-comment the relevant lines in main.

(3) Don't attempt to write all the functions at once. Here is a good order for implementing the public interface functions:

1. Implement the printError function and test it with a simple main program that just calls it with several different values for argument.
2. Implement the constructor, destructor, and print functions for **SYMBOL** next. You need the constructor function to completely create the object, and you can use the print function to verify that your next batch of functions are working correctly.
3. Add the remaining two functions to the **SYMBOL** class and test.

4. Implement and test the constructor, destructor and print functions for the **symbolList** class. Of courses, the print function can't fully be tested until you can add data into the list, but at least you can test if it prints an appropriate message when the list is empty.
5. Implement and test the addSymbol function, including testing for when the list is full.
6. Implement the remaining functions for the **symbolList** class.

(4) You will need several functions from the string library declared in string.h. Here are the prototypes and descriptions.

int strcmp(char *, char *);

Returns –1 if the first argument is alphabetically before the second argument, 0 if the two arguments are equal, and 1 if the second argument is alphabetically after the first argument.

int strlen(char *);

Returns the number of characters in the string pointed to by the argument, excluding the null character at the end of the string. NOTE: That means when you malloc space for a name in the **SYMBOL** constructor, you need to malloc strlen(argument)+1 bytes.

void strcpy(char **destination*, char **source*);

Copies the characters from *source* to the location pointed to by *destination*, excluding the null character at the end of the string. Note that *source* is the second argument and *destination* is the first argument. NOTE: strcpy may or may not copy the null character at the end, so it is a safe practice to set the last element of the area to 0 with a separate line of C++ code.

void memcpy(pointer, pointer, int);

Similar to strcpy, but the number of bytes copied is given by the third argument.

These functions are declared in the C++ built-in header file string.h.

(6) The built-in C++ function **sizeof** can be used to get the number of bytes in an object. For example, sizeof(SYMBOL) is the number of bytes in a **SYMBOL** object. This will be useful in the **getSymbol** function in the **symbolList** class. NOTE: If the argument of **sizeof** is a pointer, the value returned is the size of a pointer (typically 4), NOT the size of the object pointed to by the pointer.