CSC 539 REVIEW SHEET: C++ Classes & Libraries

- `string` class `#include <string>`

  construct with no arguments, e.g., `string str1, str2;`

  member functions include:

```
int length();           // returns number of chars in string
char at(int index);     // returns character at specified index
                        // note: indexes start at 0
int find(string substring, int startpos = 0);
                        // returns position of first occurrence of
                        // substring, starting at startpos
                        // returns string::npos if not found
int find(char ch, int startpos = 0);
                        // similarly finds a character
string substr(int startpos, int size);
                        // returns substring starting at startpos,
                        // of length size
void insert(int startpos, string substring);
                        // inserts substring at startpos position
void delete(int startpos, int size);
                        // removes substring starting at startpos,
                        // of length size
char[] c_str();         // returns the underlying C-style string,
                        // needed for library routines/classes that
                        // require C-style strings (e.g., ifstream)
```

  operators include:

```
+                       // concatenation
=                       // assignment
== != < <= > >=         // relational operators (dictionary ordering)
```

EXAMPLE CODE:

```
string str1 = "foobar";                         // DECLARES & ASSIGNS str1
str1 = str1 + "oo";                             // SETS str1 = "foobaroo"
cout << str1.length() << endl;                  // OUTPUTS 8

for (int i = 0; i < str1.length(); i++) {       // DISPLAYS "ooraboof"
    cout << str1.at(str1.length()-i-1);         // (THE STRING REVERSED)
}
cout << endl;

cout << str1.find("oo") << endl;                // OUTPUTS 1
cout << str1.find("oo", 3) << endl;             // OUTPUTS 6
cout << str1.find("ooo") << endl;               // OUTPUTS string::npos
                                                //   (which is usually MAX_INT)
string str2;
cin >> str2;
if (str1.find(str2) < str1.length()) {          // DETERMINES IF str2 IS A
    cout << str1 << " contains " << str2 << endl; // SUBSTRING OF str1
}

str1 = str1.substr(0, 6);                       // SETS str1 = "foobar"

str1.insert(3, "lish");                         // SETS str1 = "foolishbar"

str1.erase(3, 4);                               // SETS str1 = "foobar"
```

- `vector` class (templated) `#include <vector>`

  construct with one argument, the vector size e.g., `vector<int> nums(100);`
  if no size is specified, 0 items will be allocated `vector<int> empty;`

  member functions include:

```
int size();       // returns size of vector
void resize(int newsize); // resizes the vector to newsize,
                          // copying any old entries

void push_back(const <TYPE> & item);
                        // adds item to end of vector
void pop_back();        // removes item at end of vector
<TYPE> & back();        // returns item at end of vector
```

  operators include:

```
[]                      // indexing (note: first item at index 0)
=                       // assignment
```

Note: `push_back`, `pop_back`, and `back` provide the three standard operations of a stack (push, pop, and top, respectively). Thus, you can use a vector when you want the behavior of a stack.

Also: `push_back` is useful when you want to read in a sequence of values and store them as you go. You can start with an empty vector, then repeatedly add to the end using `push_back`. Once values have been read in, you can access the elements using `[]`.

EXAMPLE CODE:

```
vector<int> nums(5);                            // DECLARES VECTOR OF 5 INTS
for (int i = 0; i < nums.size(); i++) {         // STORES SQUARES IN VECTOR
    nums[i] = i*i;                              // (0, 1, 4, 9, 16)
}

nums.resize(10);                                // RESIZES TO STORE 10 INTS
for (int j = 5, j < nums.size(); j++) {         // STORES SQUARES IN NEW SPOTS
    nums[j] = j*j;                              // (25, 36, 49, 64, 81)
}

int sum = 0;
for (int k = 0; k < nums.size(); k++) {         // SUMS UP ALL 10 SQUARES
    sum += nums[k];
}


vector<string> words;                           // CREATES EMPTY VECTOR
string input;
cin >> input;
while (input != "DONE") {                       // REPEATEDLY READS IN INPUT
    words.push_back(input);                     // AND ADDS TO END OF VECTOR
    cin >> input;                               // UNTIL "DONE" IS READ
}

for (int n = 0; n < words.size(); n++) {        // DISPLAYS ALL WORDS IN VECTOR
    cout << words[n] << endl;
}
```

```
// Demonstration program to show the use of strings and vectors.
//
// The program reads in strings and displays statistics, including the
// total number of words, total number of unique words, and longest and
// shortest word lengths.  It terminates on the END-OF-INPUT character
// (^Z for Windows, ^D for UNIX).
//
// Author:  Dave Reed
////////////////////////////////////////////////////////////////////

#include <iostream>                                  // NEEDED for cin, cout
#include <string>                                    // NEEDED for string
#include <vector>                                    // NEEDED for vector
using namespace std;

int main()
{
   int totalWords = 0;
   int longest = 0, shortest = 10000;                // ASSUMES NO STRING > 10000 CHARS
   vector<string> unique;                            // CREATES DEFAULT VECTOR, SIZE 0

    string str;                                      // READ STRINGS (DELIMITED BY
    while (cin >> str) {                             //   WHITESPACE) UNTIL ^Z
          totalWords++;                              // UPDATE TOTAL # OF WORDS

        if (str.length() > longest) {                // UPDATE LONGEST WORD IF NEC.
              longest = str.length();
          }
          if (str.length() < shortest) {             // UPDATE SHORTEST WORD IF NEC.
              shortest = str.length();
          }

        bool found = false;                          // SEARCH FOR WORD IN
          for (int i = 0; i < unique.size(); i++) {//   VECTOR OF UNIQUE WORDS
              if (unique[i] == str) {                //   SO FAR.
                    found = true;                    // SET FLAG AND EXIT LOOP
                    break;                           //   IF FOUND.
              }
          }
          if (!found) {                              // IF NOT FOUND,
              unique.push_back(str);                 //   ADD TO END OF VECTOR
          }
   }

   cout << "Total number of words  = " << totalWords << endl;
   cout << "Number of unique words = " << unique.size() << endl;
   if (totalWords > 0) {
          cout << "The shortest word was length " << shortest << endl;
          cout << "The longest word was length  " << longest << endl;
   }

    return 0;
}
```

- `ifstream` and `ofstream` classes `#include <fstream>`

  `ifstream` is an input file stream (for reading data from a file)
  `ofstream` is an output file stream (for writing data to a file)

  construct with one argument, a C-style string, e.g.,

```
ifstream instr("foo.in");          ofstream ostr("foo.out");

string infile = "foo.in";          string outfile = "foo.out";
ifstream instr(infile.c_str());    ofstream outstr(outfile.c_str());
```

  can read from/write to files streams just like standard streams

```
instr >> x;                        outstr << "Howdy" << endl;
```

  when done with an fstream, should close it

```
instr.close();                     ostr.close();
```

  note: the `getline` function takes an input stream and a string variable as arguments, and reads an entire line of text from that file into the string, e.g.,

```
string line1, line2;
getline(cin, line);          // reads line1 from standard input
getline(instr, line2);       // reads line2 from file
```

```
// Demonstration program to show the use of input and output streams to
// read from and write to files.
//
// The program reads in lines of text from an input file (foo.txt) and
// echoes the lines to an output file (copy.txt), effectively copying
// the file one line at a time.
//
// Author:  Dave Reed

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    ifstream istr("foo.txt");  // OPEN INPUT STREAM, CONNECTED TO foo.txt
    ofstream ostr("copy.txt");    // OPEN OUTPUT STREAM, CONNECTED TO copy.txt

    string line;
    while (getline(istr, line)) { // REPEATEDLY READ LINE OF TEXT FROM FILE
        ostr << line << endl;     // ECHO LINE TO OUTPUT FILE (WITH endl)
    }

    istr.close();                 // CLOSE INPUT STREAM
    ostr.close();                 // CLOSE OUTPUT STREAM

    return 0;
}
```

- `istringstream` class `#include <sstream>`

  `istringstream` is an input string stream (useful for processing line-oriented text)

  construct with one argument, a C-style string, e.g.,

  ```
  istringstream instr("foo bar bizbaz");
  ```

  can read from string streams just like standard streams

  ```
  instr >> num1 >> num2;
  ```

```
// Demonstration program to show the use of input string streams to
// read and process line-oriented data.
//
// The program reads in lines of integers from the file nums.dat
// and displays the average of the integers on each line.
//
// Author:  Dave Reed
////////////////////////////////////////////////////////////////////

#include <iostream>                  // NEEDED FOR cin
#include <fstream>                   // NEEDED FOR ifstream
#include <sstream>                   // NEEDED for istringstream
#include <string>                    // NEEDED for string
using namespace std;

int main()
{
    ifstream infile("nums.dat");      // OPEN INPUT STREAM

    string str;
    while ( getline(infile, str) ) {  // REPEATEDLY READ LINE FROM FILE,
        istringstream istr(str.c_str()); // CREATE INPUT STRING STREAM TO
                                     // HOLD STRING CONTENTS (NOTE:
                                     // MUST CONVERT C++ STRING INTO
                                     // C-STYLE STRING)

        int num, count = 0, sum = 0;
        while (istr >> num) {        // CAN READ FROM INPUT STRING
            sum += num;              // JUST LIKE ANY OTHER STREAM
            count++;
        }

        if (count == 0) {
            cout << "There were no numbers to average." << endl;
        }
        else {
            cout << "The average of the " << count << " numbers is "
                 << (double)sum/count << endl;
        }
    }

    return 0;
}
```

`cctype` `library of` routines for manipulating and testing characters

`bool isalpha(ch)`	returns true if ch is alphanumeric
`bool isupper(ch)`	returns true if ch is an upper case letter ('A'..'Z')
`bool islower(ch)`	returns true if ch is a lower case letter ('a'..'z')
`bool isdigit(ch)`	returns true if ch is a digit ('0'..'9')
`bool isspace(ch)`	returns true if ch is whitespace (space, tab, CR)
`bool isspunct(ch)`	returns true if ch is a punctuation mark
`char toupper(ch)`	returns upper case version of ch (or just ch if not a letter)
`char tolower(ch)`	returns lower case version of ch (or just ch if not a letter)

`iomanip` `library of` I/O manipulators

`setiosflags(ios::fixed)`	sets flags so that real values are displayed with a fixed number of digits
`setprecision(N)`	sets flags so that real values are displayed with N digits to the right of the decimal place
`setw(N)`	sets flags so that next value displayed will be right-justified in a field of N characters (useful for aligning columns)

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

int main()
{
    ifstream istr("words.txt");              // OPEN INPUT STREAM, CONNECTED TO words.txt

    int spaceCount = 0, digitCount = 0,      // INITIALIZE COUNTERS
        alphaCount = 0, totalCount = 0;

    string line;
    while (getline(istr, line)) {            // REPEATEDLY READ LINE OF TEXT FROM FILE
        for (int i = 0; i < line.length(); i++) {  // FOR EACH CHARACTER IN THE LINE,
            if (isalpha(line.at(i))) {             //   IF IT'S A LETTER, INCR. COUNT
                alphaCount++;                      //
            }                                      //
            else if (isdigit(line.at(i))) {        //   IF IT'S A DIGIT, INCR. COUNT
                digitCount++;                      //
            }                                      //
            else if (isspace(line.at(i))) {        //   IF IT'S A SPACE, INCR. COUNT
                spaceCount++;                      //
            }                                      //
        }
        totalCount += line.length();         // INCREMENT TOTAL CHAR COUNT
    }

    istr.close();                            // CLOSE INPUT STREAM

    cout << setiosflags(ios::fixed) << setprecision(1);  // FIX TO 1 DIGIT TO RIGHT
                                                         // OF DECIMAL PLACE
    cout << "Total # of chars: " << totalCount << endl;
    cout << "  # of letters: " << setw(5) << alphaCount  // DISPLAY ALIGNED COUNTS
         << "  (" << (alphaCount * 100.0)/totalCount << "%)" << endl;
    cout << "  # of digits : " << setw(5) << digitCount
         << "  (" << (digitCount * 100.0)/totalCount << "%)" << endl;
    cout << "  # of spaces : " << setw(5) << spaceCount
         << "  (" << (spaceCount * 100.0)/totalCount << "%)" << endl;

    return 0;
}
```

- `stack` class (templated)

```
#include <stack>
```

construct with no arguments, e.g.,

```
stack<int> nums;
stack<string> words;
```

member functions include:

```
void push(const <TYPE> & item);     // adds item to top of stack
void pop();                         // removes item at top of stack
<TYPE> & top();                     // returns item at top of stack
bool empty();                       // returns true if stack is empty
int size();                         // returns size of stack
```

operators include:

```
=                                   // assignment
```

```
// Demonstration program to show the use of stacks in parsing expressions to see if parentheses
// match up.  The program reads in expressions, one per line, and displays whether the expression
// is valid (all parentheses match up) or invalid (missing or unmatched parentheses).
//
// Author: Dave Reed
////////////////////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main()
{
    string line;
    while ( getline(cin, line) ) {                  // REPEATEDLY READ IN LINE OF TEXT
        stack<char> parens;                         // CREATE STACK OF PARENTHESES

        string status = "VALID";
        for (int i = 0; i < line.length(); i++) {   // FOR EACH CHAR IN THE LINE
            if (line[i] == '(') {                   //   IF LEFT PARENTHESIS,
                parens.push(line[i]);               //     PUSH ONTO STACK
            }
            else if (line[i] == ')') {              //   OTHERWISE, IF RIGHT PARENTHESIS,
                if (parens.empty()) {               //     IF NO LEFT PAREN ON STACK,
                    status = "INVALID";             //       THEN EXPRESSION IS INVALID
                    break;
                }
                else {
                    parens.pop();                   //     OTHERWISE, POP OFF MATCHING LEFT PAREN
                }
            }
        }

        if (!parens.empty()) {                      // IF ANY LEFT PARENS REMAIN UNMATCHED,
            status = "INVALID";                     //   THEN EXPRESSION IS INVALID
        }

        cout << status << endl;                     // DISPLAY STATUS OF EXPRESSION
    }

    return 0;
}
```

- `queue` class (templated) `#include <queue>`

  construct with no arguments, e.g.,
  ```
  queue<int> nums;
  queue<string> words;
  ```

  member functions include:
  ```
  void push(const <TYPE> & item);   // adds item to back of queue
  void pop();                       // removes item at front of queue
  <TYPE> & front();                 // returns item at front of queue
  bool empty();                     // returns true if queue is empty
  int size();                       // returns size of queue
  ```

  operators include:
  ```
  =                                 // assignment
  ```

- `priority_queue` class (templated) `#include <queue>`

  a priority queue can store any type for which the < operator is defined

  construct with no arguments, e.g.,
  ```
  priority_queue<int> nums;
  priority_queue <string> words;
  ```

  member functions include:
  ```
  void push(const <TYPE> & item);   // adds item to priority queue
  void pop();                       // removes item with highest priority
  <TYPE> & top();                   // returns item with highest priority
  bool empty();                     // returns true if empty
  int size();                       // returns size of priority_queue
  ```

  operators include:
  ```
  =                                 // assignment
  ```

EXAMPLE CODE:

```
// QUEUES ARE FIRST-IN-FIRST-OUT LISTS

queue<string> words;                        // DECLARES QUEUE OF WORDS

words.push("foo");                          // PUSH A SERIES OF WORDS
words.push("bar");
words.push("biz");
words.push("baz");

while (!words.empty()) {                    // WILL DISPLAY THE SEQUENCE:
    cout << words.front() << endl;          //   foo, bar, biz, baz
    words.pop();
}


// PRIORITY QUEUES ARE ORDERED LISTS, PROVIDE ACCESS TO LARGEST ITEM

priority_queue<string> ordered;             // DECLARES QUEUE OF ORDERED WORDS

ordered.push("foo");                        // PUSH A SERIES OF WORDS
ordered.push("bar");
ordered.push("biz");
ordered.push("baz");

while (!ordered.empty()) {                  // WILL DISPLAY THE SEQUENCE:
    cout << ordered.top() << endl;          //   bar, baz, biz, foo
    ordered.pop();
}
```

- `set` class (templated) `#include <set>`

  stores a collection of unique items, multiple insertions of same item result in only one entry

  construct with no arguments, e.g., `set<int> nums;`
  `set<string> words;`

  member functions include:

```
void insert(const <TYPE> & item);   // adds item to set (no duplicates)
void erase(const <KEY_TYPE> & key); // removes item from the set
<set>::iterator find(const <TYPE> & item); // returns whether item in set
bool empty();                       // returns true if set is empty
int size();                         // returns size of set
```

  operators include:

```
=                                   // assignment
```

  The `find` member function returns an iterator, pointing to where the item was found.
  If the item was not found, returns same as `end()`, e.g., `nums.find(10) == nums.end()`

  To traverse the contents of a set, use an iterator (which must be dereferenced to access each successive element), e.g.,

```
for (set<string>::iterator iter = nums.begin(); iter != nums.end(); iter++) {
    cout << *iter << endl;
}
```

- `map` class (templated) `#include <map>`

  stores a collection of entries, with associated keys

  construct with no arguments, e.g., `map<string, int> wordCount;`
  (template types refer to key and data, respectively)

  member functions include:

```
<ENTRY_TYPE> & operator[](const <KEY_TYPE> & key);
                                    // accesses the map entry with key
void erase(const <KEY_TYPE> & key); // removes entry from the map
<map>::iterator find(const <KEY_TYPE> & key); // returns whether in map
bool empty();                       // returns true if empty
int size();                         // returns size of map
```

  operators include:

```
=                                   // assignment
```

  The `find` member function returns an iterator, pointing to where the item was found.
  If the item was not found, returns same as `end()`, e.g.,
  `wordCount.find("foo") == wordCount.end()`

  To traverse the contents of a set, use an iterator, which must be dereferenced to access a struct containing the key (accessed as `first`) and the entry (accessed as `second`), e.g.,

```
for (map<string,int>::iterator iter=wordCount.begin(); iter!=wordCount.end(); iter++) {
    cout << iter->first << ": " << iter->second << endl;
}
```

```
// Demonstration program to show the use of sets and maps to store words and their frequency.
// As each word is read in, it is inserted into a set of strings.  Thus, this set maintains a
// list of all of the words seen so far (with no duplicates).  Likewise, a separate map is
// maintained that stores a count for each word.  The key for each entry is the word itself,
// and the entry is the number of occurrences of that word.
//
// Author: Dave Reed
////////////////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;


int main()
{
    map<string, int> wordCount;                             // MAP OF WORDS & THEIR COUNTS
    set<string> uniqueWords;                                // SET OF UNIQUE WORDS

    string str;
    while (cin >> str) {                                    // REPEATEDLY READ NEXT str
        if (wordCount.find(str) == wordCount.end()) {       //   IF ENTRY NOT FOUND FOR str,
            wordCount[str] = 1;                             //     ASSIGN NEW ENTRY WITH VALUE 1
        }
        else {
            wordCount[str]++;                               //     OTHERWISE, INCREMENT ENTRY
        }

        uniqueWords.insert(str);                            //   ADD WORD TO SET OF ALL WORDS
    }

                                                            // TRAVERSE AND DISPLAY THE MAP ENTRIES
    for (map<string,int>::iterator iter = wordCount.begin(); iter != wordCount.end(); iter++) {
        cout << iter->first << ": " << iter->second << endl;
    }

                                                            // TRAVERSE AND DISPLAY THE SET ELEMENTS
    for (set<string>::iterator iter = uniqueWords.begin(); iter != uniqueWords.end(); iter++) {
        cout << *iter << endl;
    }

    return 0;
}
```