CSCI-1200 Data Structures — Spring 2016
Test 2 — Solutions

1 Simple Line Editor [/25]

This question involves a simple line oriented text editor. This type of editor was common before the days of high speed data connections and was used with printing terminals. Line oriented editors manipulate text one line at a time.

Commands are entered from the keyboard. Editor commands are single letters. The editor keeps track of the *current line* at all times. Most commands operate on the current line. A few will operate on the whole file.

Text will be stored in an STL list of strings. The current line should be indicted by an STL list iterator.

The editor supports the following commands:

```
N - Advance the current line pointer forward one line.
    If advancing would move the current line pointer beyond the end of the text, keep
    the pointer on the last line.
P - Move the current line pointer to the previous line.
    If advancing would move the current line pointer beyond the first line of the text, keep
    the pointer on the first line.
L - List (print) the current line. The current line does not change;
D - Delete the current line. The current line is the line after the line deleted.
E - Erase. Delete all lines. The current line is the end iterator.
S - Search forward for a text string in a line starting at the current line.
    The search text is entered on the next line after the command.
    The current line pointer is moved to the line containing the search text.
    If the text is not found, the current line pointer is unchanged.
Q - Quit Exit the program.
I - Insert text. The text line to be inserted is entered on the line following the command.
    The new text is inserted after the current line. The current line pointer is moved to the inserted line.
R - Replace. Replace the text of the current line. The new text is enetered on the next input line.
    The entire line is replaced with the new text. The current line pointer is not moved.
```

1.1 `Line Editor` Private Variables [/4]

The editor class stores the the text lines as an STL list of strings and the current line pointer as a list iterator.

Write the private variables for the Editor class as they would appear in `editor.h`. You don't have to write the rest of class definition.

Don't worry about `#include` or `#define` statements.

Solution:

```
  std::list<std::string> lines;
  std::list<std::string>::iterator current_line;
  int line_count; // optional - if used should be calculated correctly in part 2
```

1.2 `Editor` Class Implementation [/16]

Now implement the constructor, member functions, and any helper functions needed to implement the commands as they would appear in the `.cpp` file. Write a function for each command except quit. Also write a constructor if needed. Include a comment line for each function to indicate which command the method implements. Be sure to update the current line iterator when necessary.

Solution:

```
// constructor
Editor::Editor() {
  current_line = lines.end();
}

// N - advance
```

```
void Editor::forward_line() {
  if (current_line != lines.end())
    current_line++;
  if (current_line == lines.end())
    current_line--;
}

// P - previous line
void Editor::backward_line() {
  if (current_line == lines.end())
    return;
  if (current_line != lines.begin())
    current_line--;
}

// L - list (print)
void Editor::print() const {
  if (current_line == lines.end())
    return;

  std::cout << *current_line << std::endl;
}

// D - delete line
void Editor::erase() {
  if (current_line == lines.end())
    return;

  current_line = lines.erase(current_line);

  if (current_line == lines.end() && lines.size() > 0)
    --current_line;
}

// E - erase all lines
void Editor::erase_list() {
  lines.clear();
  current_line = lines.end();
}

// S - search
void Editor::search(const std::string& str) {
  std::list<std::string>::iterator p = current_line;
  while (p != lines.end()) {
    if ((*p).find(str) != std::string::npos) // npos = -1
      break;
    ++p;
  }

  if (p != lines.end())
    current_line = p;
}

// I - insert
void Editor::insert(const std::string& str) {
  if (current_line == lines.end()) {
    lines.push_back(str);
    --current_line;
  }
  else {
    ++current_line;
    current_line = lines.insert(current_line, str);
  }
}
```

```
// R - replace line
void Editor::replace(const std::string& str) {
  *current_line = str;
}
```

1.3 `Editor` Main Program Implementation [/5]

The main program should read a file name from the command line, read lines from the file and insert them into the editor. If the command line does not contain a file name, write code to display a message indicating that a file name is required. Write code to check to see if the file can actually be read. Do not write the entire main routine, just the input checking code.

Solution:

```
int main(int argc, char* argv[]) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " file_name" << std::endl;
    return 1;
  }

  std::ifstream in_file(argv[1]);
  if (! in_file.good()) {
    std::cerr << "Can't open " << argv[1] << std::endl;
    return 1;
  }

...
return 0;
}
```

2 Fibonacci Numbers and Memoization [/20]

In mathematics, the Fibonacci numbers or Fibonacci sequence are the numbers in the following integer sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

In mathematical terms,

$F_0 = 0$;
$F_1 = 1$;
$F_n = F_{n-1} + F_{n-2}$ for all $n > 1$.

2.1 `Recursive Fibonacci` Function [/5]

Write a simple *recursive* function to calculate F_n. The function should be passed the value of n as an unsigned int and return the value of F_n as an unsigned int.

Solution:

```
unsigned int fib(unsigned int n) {
  if (n <= 1)
     return n;

  return fib(n-1) + fib(n-2);
}
```

2.2 `Iterative Fibonacci` Function [/5]

Now, rewrite the Fibonacci calculation as an iterative function.

Solution:

```
unsigned int fib(unsigned int n) {
  if (n <= 1)
     return n;

  unsigned int f2 = 0;
```

```
  unsigned int f1 = 1;
  unsigned int result = 0;
  for (unsigned int i = 2; i <= n; ++i) {
    result = f1 + f2;
    f2 = f1;
    f1 = result;
  }

  return result;
}
```

2.3 `Recursive Fibonacci` Function with Memoization [/10]

One problem with the recursive method of calculating F_n is that we are solving the same subproblem many times. In order to find F_{n-1}, we must find F_{n-2}, but we then again solve for F_{n-2} to calculate the Fibonacci sum. Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.

Rewrite your recursive Fibonacci calculation using an STL map with an unsigned int as a key and an unsigned int as a value to store already calculated values of F_n. If the value of F_n is in the map, return it. If not, it should calculate and store the value for F_n.

Solution:

```
unsigned int fib(unsigned int n) {
  static std::map<unsigned int, unsigned int> memo;

  if (n <= 1)
    return n;

  if (memo.find(n) != memo.end())
    return memo[n];

  unsigned int result = fib(n-1) + fib(n-2);
  memo[n] = result;

  return result;
}
// pass-by-reference version:
unsigned int fib(unsigned int n, std::map<unsigned int, unsigned int> &memo) { ... }
```

3 Matrix Addition [/18]

The sum of two matrices A and B is defined as $c_{ij} = a_{ij} + b_{ij}$. That is, we calculate the sum by performing an element by element addition of the two matrices.

3.1 `Matrix` Class [/10]

Write a templated matrix class to contain the matrix data as it would appear in class.h. The constructor should be passed the number of rows and columns. Use accessors to retrieve the the row and column sizes, and individual elements of the matrix. Use a modifier to set element values. Store the matrix data in an STL vector of vectors. You don't have to overload any operators in the class. Be sure to use const where appropriate. One line methods should be included in the class definition.

Solution:

```
template<class T>
class Matrix {
public:
  Matrix(int rows, int cols);

  int getRows() const {return rows_;}
  int getCols() const {return cols_;}
  T get_element(int i, int j) const {return data[i][j];}

  void set_element(int i, int j, const T& value) {data[i][j] = value;}
```

```
private:
  int rows_;
  int cols_;
  std::vector<std::vector<T> > data;
};


template <class T>
Matrix<T>::Matrix(int rows, int cols) {
  rows_ = rows;
  cols_ = cols;
  data.resize(rows_, std::vector<T>(cols_, 0));
}
```

3.2 `Matrix` Addition Operator [/8]

Write an overloaded helper + operator to add two matrices of the templated class and return a new matrix containing the sum. You function should check that both matrices are the same size and print an error message and exit if they are not.

Solution:

```
template <class T>
Matrix<T> operator+(const Matrix<T>& A, const Matrix<T>& B) {
  assert(A.getRows() == B.getRows());
  assert(A.getCols() == B.getCols());

  Matrix<T> C(A.getRows(), A.getCols());
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      C.set_element(i, j, A.get_element(i,j) + B.get_element(i,j));
    }
  }

  return C;
}
```

4 Short Answers [/ 20]

4.1 Debugging tools [/ 4]

In what circumstances is using a debugging tool such as Dr. Memory or valgrind a better solution than adding print statements to your code?

Solution: The debugger is more useful for finding memory related errors such as memory leaks, or segmentation faults caused by bad address access. The debugger can be used to set breakpoints, watchpoints, examine the call stack, and step through the program while examining variables' values.

4.2 Map Search [/ 4]

How do I know if an std::map insert actually inserted a new item into the map? What does map insert return?

Solution: The STL map insert method returns a pair. The first element of the pair is an iterator to the newly inserted pair or to the pair that already existed in the map. The pair's second element is set to true if a new element was inserted or false if an equivalent key already existed. In the case of false, the value is not changed.

4.3 List Insertion [/ 4]

What is the order of insert operations into the middle of an STL vector? Into the middle of an STL list? Is an iterator that refers to an object in the vector invalidated? Why or why not? A list iterator, why or why not?

Solution: For an STL vector, it is O(n). For an STL list, it is O(1). Iterators that refer to elements within a vector become invalid because the data may be moved or reallocated. A list iterator is still valid.

4.4 Pointers [/ 4]

The following code contains an error, describe the error.

```
class Node {
public:
  Node(int v) { prev=next=NULL; value=v; }
  Node *prev;
  Node *next;
  int value;
};

int main() {
  Node *tmp = new Node(7).
  /* some initialization and insertion of other nodes */
  tmp->prev->next = tmp->next;
  delete tmp;
  tmp->next->prev = tmp->prev;

return 0;
}
```

Solution: After delete, tmp no longer contains a valid memory address.

4.5 Pairs [/ 4]

Find the error in the following code

```
std::pair<int, double> p1(5, 7.5);
std::pair<int, double> p2 = std::make_pair(8, 9.5);
p1.first = p2.first;
std::pair<const std::string, double> p3 = std::make_pair(std::string("hello"), 3.5);
p3.second = -1.5;
p3.first = std::string("good bye");
```

Solution: The first element in p3 is declared const. It cannot be modified.

5 iClicker Review [/ 14]

5.1 Which of the following statements is *false*?

Solution: C

(A) When you use the STL sort routine to organize a collection of integers, small values will be at the front, big values at the end.

(B) I can store multiple copies of the same value in an STL `vector` and if that vector is then sorted, those duplicates will still be there, just clustered together.

(C) If you have a big dataset and you care about performance, you should write your own sort routine because the STL `vector` sort routine is inefficient.

(D) The STL sort routine can alphabetize STL `string`s.

(E) The STL sort requires use of something called an `iterator`, and apparently we are expected to use them without knowing much of anything about them.

5.2 Which of the following is *true* for the STL map iterators?

Solution: E

(A) Data is accessed in the order it was inserted.

(B) Visiting every element in an STL map is faster than visiting every element in an STL vector.

(C) Since STL map has an `operator[]` it is like STL vector and I move a map iterator forward not just one spot (using `itr++`), but I can also jump forward and arbitrary number of spots (e.g., 25) using `itr + 25`.

(D) If a map iterator is dereferenced it simply returns the second element of the pair referenced by the iterator.

(E) None of the above.

5.3 Which of the following statements is *not true* about memory debuggers?

(A) Even though modern computers have an obscene amount of RAM and hard drive space, it is still important to use a memory debugger to ensure all dynamically-allocated memory has been deleted.

(B) A memory debugger points to the line number in the code that must be edited to fix the memory leak.

Solution: B

(C) Dr. Memory and/or Valgrind are available on Linux, MacOSX, and Windows platforms.

(D) Code that runs perfectly on a student's computer may still have a memory error.

(E) Step-by-step debuggers (like `gdb`, `lldb`, and your IDE debugger) and memory debuggers (like Dr. Memory and Valgrind) are complementary. They both play a significant role in the debugging process.

5.4 Which of the following statements about STL vectors and STL lists is *not true*?

(A) The elements in both an STL vector and an STL list can be sorted efficiently, in $O(n\ log\ n)$ time.

(B) Both vectors and lists have the `push_back` and `pop_back` operations, but only list has `push_front` and `pop_front`.

Solution: C

(C) STL vector and STL list iterators are interchangeable.

(D) Both structures are dynamically resizeable depending on the quantity of data necessary to be stored.

(E) Both vector and list have an `erase` member function, but the list version of the function is constant-time while the vector function has linear running time.

5.5 What is the order notation of the running time and space/memory for the merge sort function we implemented in lecture?

(A) running time: $O(n\ log\ n)$ space/memory: $O(log\ n)$

(B) running time: $O(n)$ space/memory: $O(n\ log\ n)$

Solution: E

(C) running time: $O(n\ log\ n)$ space/memory: $O(n\ log\ n)$

(D) running time: $O(n)$ space/memory: $O(log\ n)$

(E) running time: $O(n\ log\ n)$ space/memory: $O(n)$

5.6 Why are the `operator<` and `operator>` comparison operators not available for iterators of the STL `list` or our `dslist` class?

(A) These operators cannot be implemented efficiently because list memory is not arranged contiguously like it is for vectors.

Solution: A

(B) There is never a need for these operators since we have `operator==` and `operator!=` defined for list iterators.

(C) These operators can only be implemented for data values of pointer type.

(D) The meaning of these operators would be unclear or ambiguous to the users of these classes.

(E) All of the above.

5.7 If a class (templated or non-templated) has member variables that are dynamically-allocated (on the heap!), which of the following functions must be custom written (the default version provided by the compiler is insufficient)?

(A) The copy constructor

(B) The assignment operator

Solution: D

(C) The destructor

(D) All of the above

(E) None of the above