COMP 151H: Object Oriented Programming (Honor Track)
Spring Semester 2007
Midterm Exam
Instructor: Chi Keung Tang
Monday, March 19, 2007
7:00 – 9:00pm
Room 3007

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of four (4) problems. Follow the instructions carefully. Please write legibly in the space provided. Keep the exam booklet stapled.

K E Y

Problem	Points	your score
1 HASH TABLE AND TEMPLATES	5	
2 `static` AND `const` MEMBERS	3	
3 POINTERS AND CLASS POINTERS	3	
4 INTERFACE VS IMPLEMENTATION	4	
Total	15	

1 Hash Table and Templates

Assume the template class *DoublyLinkedList* is already defined (in `Lish.h`, shown below) and implemented (in `List.cpp`).

```
template<class ElemType>       // ElemType must define the operator<
class DoublyLinkedList
{
    public:
        DoublyLinkedList();
        ~DoublyLinkedList();
        DoublyLinkedList(const DoublyLinkedList& copy);
        DoublyLinkedList& operator=(const DoublyLinkedList& assign);

        // insert an element
        void insert(const ElemType& element);

        // clear the whole list
        void clear();

        // other member functions ...

    private:
        // other members ...
};
```

Notice: You **must** answer this question by making use of the above definition. Do not define your own linked list and hash table. However, you are free to use any member functions defined in class *DoublyLinkedList*.
The template class *HashTable* is also defined in `HashTable.h`.

```
#include "List.h"

template<class ElemType>        // ElemType must define the operator<
class HashTable
{
    public:
        typedef unsigned int (*HashFunc) (const ElemType&);

        // hashFunc must return an integer in-between 0 and (numBuckets-1)
        HashTable(int numBuckets, HashFunc hashFunc);   // (1)

        HashTable(const HashTable& copy);               // (2)
        HashTable& operator=(const HashTable& assign);  // (3)
        ~HashTable();

        // insert an element
        void insert(const ElemType& element);

        // other member functions

    private:
        DoublyLinkedList<ElemType>* m_buckets;  // buckets
        unsigned int    m_numBuckets;           // number of buckets
        HashFunc        m_hashFunction;         // hash function
};
```

Notice: You **must** answer this question by making use of the above definition. Do not define your own linked list and hash table. However, you are free to use any member functions defined in class *DoublyLinkedList*.

(a) Define the following member functions

```
HashTable(int numBuckets, HashFunc hashFunc);
HashTable(const HashTable& copy);
HashTable& operator=(const HashTable& assign);
```

Solution:

```
//  Note: CheckSufficientMemory() and SAFE_DEL_ARRAY() are helper functions
//  and is clear from context.
//  Students answering this question should implement this in answering this question,
//  which is only a simple if-else.

template<class ElemType>
HashTable<ElemType>::HashTable<ElemType>(int numBuckets, HashFunc hashFunc)
{
m_buckets = new DoublyLinkedList<ElemType>[numBuckets];
CheckSufficientMemory(m_buckets);

m_numBuckets = numBuckets;
m_size = 0;
m_hashFunction = hashFunc;
}

template<class ElemType>
HashTable<ElemType>::HashTable<ElemType>(const HashTable<ElemType>& copy)
{
  m_numBuckets = copy.m_numBuckets;
  m_buckets = new DoublyLinkedList<ElemType>[m_numBuckets];
  CheckSufficientMemory(m_buckets);

  for (unsigned int i=0; i<m_numBuckets; i++)
    m_buckets[i] = copy.m_buckets[i];

  m_size = copy.m_size;
  m_hashFunction = copy.m_hashFunction;
}

template<class ElemType>
HashTable<ElemType>& HashTable<ElemType>::operator=(const HashTable<ElemType>& assign)
{
  if ( this != &assign )
  {
    // clear the old hash table
    SAFE_DEL_ARRAY(m_buckets);

    m_numBuckets = assign.m_numBuckets;
    m_buckets = new DoublyLinkedList<ElemType>[m_numBuckets];
    CheckSufficientMemory(m_buckets);

    for (unsigned int i=0; i<m_numBuckets; i++)
      m_buckets[i] = assign.m_buckets[i];

    m_size = assign.m_size;
    m_hashFunction = assign.m_hashFunction;
   }

   return (*this);
}
```

(b) Discuss the pros and cons of template in generic programming.

Solution:

Pros: Programming code up and maintain only one version of function/class.
(static polymorphism)

Cons: Should be used with care; otherwise an innocent looking program may
result in a large executable.

2 `static` and `const` members

(a) Name and explain two advantages of using a static data member over using a global object

```
Solution:
(1) A static member is not entered into the program's global
    namespace, thus removing the possibility of an accident
    conflict or unintentional changes.
(2) Information hiding can be enforced. A static member can
    be a private member, a global object cannot.
```

(b) Add a `static` class member `double m_interestRate` and initialize it to `0.0589`.

```
Solution:

class Account {
public:
  Account (double amount, const string &owner);
  string owner () const { return m_owner; }
private:
  static double _interestRate;                 // declaration here
  double m_amount;
  strong m_owner;
};

double Account::m_interestRate = 0.0589;       // initialization here
```

(c) Name one reason for using *const* data member.

Solution: The value of the data member will not be changed after the object has been constructed.

(d) Add a `const` class member `Date m_birthdate` and initialize it. Assume the constructor `Date (int, int, int)` is available.

```
Solution:

class Employee {
public:
  Employee (int yy=0, int mm=0, int dd=0);
private:
  const Date m_birthdate;                 // define const member here
  // ... other data member
};

Employee::Employee (int yy, int mm, int dd)
  : m_birthdate (yy, mm, dd)              // member initialization
{
  // other initialization
}
```

3 Pointers and Class Pointers

(a) Given a nonmember (global) function `HeightIs()`:

```
int HeightIs();
```

Define a pointer to a function `pfi` that takes no parameters and has a return value type of `int` and initialize it to point at `HeightIs()`.

```
Solution:
  int (*pfi) () = HeightIs;
```

(b) Assume we have a class `Screen` which has a member function – `height()` – which also take no parameters and have a return type of `int`:

```
inline int Screen::height() { return m_height; }
```

The assignment of `height()` to `pfi`, however, is a type violation. In $C++$, a member function has an additional type attribute absent from a nonmember function – *its class*. For example, the type of the member function is:

```
int Screen::*
```

(c) Define a "pointer to member function of class `Screen` of return type `int` which takes no parameters", `pfi`, and initialize it to point at the member function `Screen::HeightIs()`.

```
Solution:
  int (Screen::*pfi)() = &Screen::HeightIs;
```

(d) Given the following class definition for `Image`:

```
class Image
{
public:
  Image (int width=0, int height=0);
  // ... other member functions
private:
  int m_width;              // width of image
  int m_height;             // height of image
  unsigned char **m_img;    // dynamic storage
};
```

Define the default constructor function. Use the member initialization list to initialize `m_width` and `m_height`, and allocate sufficient memory for a new image. Check NULL memory.

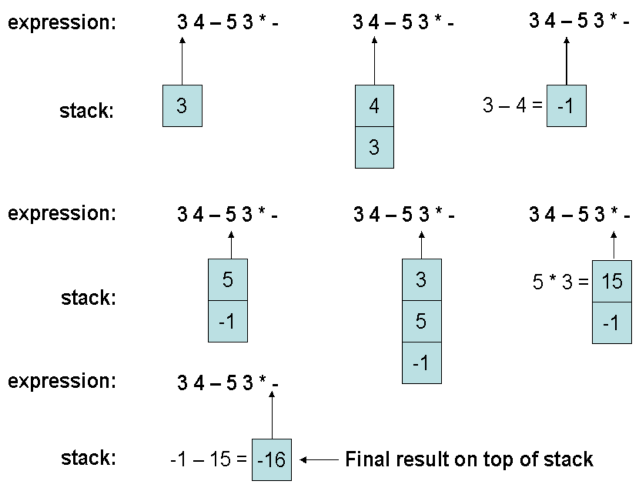
```
Solution:

Image::Image (int width, int height) : m_width(width), m_height(height)
{
  m_img = new unsigned char* [width];
  if (!m_img) {
     cerr << "error" << endl;
     exit (1);
  }
  for (int i=0; i<width; i++) {
    m_img[i]  = new unsigned char [height];
    if (!m_img[i]) {
     cerr << "error" << endl;
     exit (1);
    }
  }
}
```

4 Interface vs Implementation

Write a program to evaluate expressions in *Reverse Polish Notation (RPN)* or *postfix* notation, using the `stack` class and other classes provided for you. The evaluation rule is:

1. Evaluate the given expression from left to right.
2. At each occurrence of an operator, apply it to the two operands to the immediate left and replace the sequence of two operands and one operator by the resulting value.

An example is shown in the figure below.



To simplify the problem, we make use of the following `Token` class and the associated facilities. A tokenizer is a program that reads the input, and tokenizes the input string into a set of tokens which are of the following types:

```
enum tokenType { operandToken, operatorToken, eolToken, eofToken, badToken };
```

Each token will be one of the enumerated types. An operand will be an `operandToken`; an operator will be an `operatorToken`. The end of a line and end of the input file will be signaled by the special `eolToken` and `eofToken`. Finally, if the tokenizer cannot interpret the input, it will mark it as a `badToken`, leaving it to the client to respond appropriately.
The operators are classified using the following declaration:

```
enum operatorType { none, add, subtract, multiply, divide };
```

The `none` type is used if the client mistakenly requests the operator type for a token that's not an operator.
The code below contains the class definition that defines the public interface for the `Token` class:

```
#ifndef TOKEN_H
#define TOKEN_H

class Token {
public:
  tokenType     nextToken();   // get next token from the input stream
  operatorType  getOperator(); // return operator
  double        getOperand();  // return operand
```

```
// other declarations...
};

#endif
```

The `Stack` class provided is a class template:

```
#ifndef STACK_H
#define STACK_H

template <class ElementType>
class Stack {
public:
  Stack ();
  void push (ElementType e);
  ElementType pop();
  ElementType top();
  bool isEmpty();
private:
  // private data structures...
};

#endif
```

You must use the following skeleton and complete the program:

```
int main()
{
  Token t;
  Stack <double> s;
  double op1, op2;
  bool done(false);

  while (!done) {
    switch (t.nextToken()) {
      case operandToken:


      case operatorToken:
        // op2 is the top of the stack, op1 is the next value down
        // first, have to make sure there are two items to pop

      case eofToken:
        done = true;   // break intentionally omitted here
      case eolToken:   // if there's something in stack, display it
        if (!s.isEmpty())
          fprintf (stdout, "--> %f\n",  s.pop());
        // s.clear();  // clear stack for the next line
        break;
      case badToken:
        fprintf (stderr, "Input error!\n");
        break;
    }
  }
  return 0;
```

Solution:

```
#include "utilities.h" // enumerated types definitions
#include "stack.h"     // Stack class
#include "token.h"     // token class

int main()
{
  Token t;
  Stack <double> s;
  double op1, op2;
  bool done(false);

  while (!done) {
    switch (t.nextToken()) {
      case operandToken:
        s.push(t.getOperand());
        break;
      case operatorToken:
        // op2 is the top of the stack, op1 is the next value down
        // first, have to make sure there are two items to pop
        if ( (s.isEmpty()) || (op2 = s.pop() && s.isEmpty() ))
          fprintf (stderr, "Not enough operands for operator!\n");
        else {  // get op1, then apply appropriate operator
          op1 = s.pop();
          switch (t.getOperator()) {
            case add:      s.push (op1 + op2); break;
            case subtract: s.push (op1 - op2); break;
            case multiply: s.push (op1 * op2); break;
            case divide:
              if (op2 == 0)
                fprintf (stderr, "Division by zero!\n");
              else
                s.push (op1 / op2);
              break;
           }
        }
        break;
      case eofToken:
        done = true;   // break intentionally omitted here
      case eolToken:   // if there’s something in stack, display it
        if (!s.isEmpty())
          fprintf (stdout, "--> %f\n",  s.pop());
        // s.clear();  // clear stack for the next line
        break;
      case badToken:
        fprintf (stderr, "Input error!\n");
        break;
    }
  }
  return 0;
}
```