THE HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY
Department of Computer Science & Engineering

COMP 152: Object-Oriented Programming and Data Structures
Fall 2010

Midterm Examination

Instructor: Gary Chan

Date: Saturday, 23 October 2010
Time: 2:30pm – 4:00pm
Venue: LTC

- **This is a closed-book examination. However, you are allowed to bring with you a piece of A4-sized paper with notes written or typed on both sides for use in the examination.**
- **Your answers will be graded according to correctness, efficiency, precision, and clarity.**
- **During the examination, you must put aside your calculators, mobile phones, PDAs and all other electronic devices. In particular, all mobile phones should be turned off completely.**
- **This booklet has 21 single-sided pages. Please check that all pages are properly printed before you start working.**
- **You may use the reverse side of the pages for your rough work or continuation of work.**

Student name: _____ **English nickname (if any):** _____

Student ID: _____ **Email:** _____ **Assigned ID:** _____

I have not violated the Academic Honor Code in this examination (your signature): _____

Question	Your score	Maximum score
1		20
2		30
3		25
4		25
TOTAL		100

1. **Basic C++: Pointer, Recursion and File I/O** *(20 points total)*

   In this question, you are going to implement two functions to build and delete a ring structure, respectively.

   You are given the following `RingNode` definition which holds a `char`:

```
class RingNode{
public:
  char value;  // value stored in the node
  RingNode* next; // pointer to the next RingNode
};
```

   You are also given the following function to print the ring, given a pointer to the beginning of the ring (given by `head`) and the size of the ring (given by `size`):

```
// print the ring
// head: pointer to the beginning of the ring
// size: number of nodes in the ring
void print_the_ring(const class RingNode* head, int size){
  for (int i = 0; i < size; i++){
    cout << head->value << endl;
    head = head->next;
  }
}
```

(a) *(12 points)* Implement a *recursive* read function to build a ring given a file stream with the prototype:

```
int read_recursively (ifstream& fin, RingNode*& current);
```

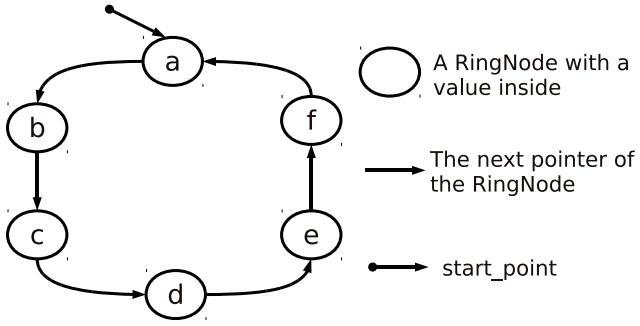
This function takes the `fin` of `ifstream` as input and reads in the first line of the file to build a ring, i.e., it reads character by character of the line until *either* end of file *or* `\n` is encountered, whichever is earlier. You can assume that `fin` is opened correctly.

The function takes in the RingNode pointer `current`, which has been initialized to `NULL`. The function returns the number of nodes in the ring, with `current` pointing to the node of the first character in the line. If there is no character before end-of-file or `\n`, then `current` should be NULL.

For example, given the following codes:

```
int main(){
  ifstream fin("input.txt");
  RingNode* start_point = NULL;
  int total = read_recursively(fin, start_point);
  print_the_ring(start_point,total);
//....
```

If the content of `input.txt` is just a line of `abcdef`, the ring should be like below:



and the output of the program by calling `print_the_ring(current, 6 )` is:

```
a
b
c
d
e
f
```

Implement the *recursive* read function below. You may use
`ifstream & ifstream::get ( char & c );`
to read character from an input stream and
`bool ifstream::eof ();`
to determine the end-of-file status.

```
// Recursive read in character by character of an input file
// to build a ring
// fin: input stream
// current: initialized to NULL and points to the first node upon exit
// returns the number of nodes in the ring
int read_recursively (ifstream& fin, RingNode*& current){
```

(b) *(8 points)* Implement a *recursive* delete function to destroy a ring with prototype

```
void delete_recursively (RingNode*& current);
```

The input parameter `current` points to the first node in the ring, and upon function exit, the whole ring is deleted with `current` pointing to `NULL`. You can assume that `current` is either `NULL` or pointing to a valid RingNode when the function is firstly invoked.

```
// Recursively delete a ring
// current: pointing to the first node or NULL
void delete_recursively (RingNode*& current){
```

2. **Basic OOP: Polygon** *(30 points total)*

A *Polygon* is characterized by m points (or sides). Each of the points is n-dimensional. For example, for point i, its coordinate is given by $(x_{i,0}, x_{i,1}, \ldots, x_{i,n-1})$.

You are to implement a container class for *Polygon* using an internal 2D array `point`. `point[i]` is the ith point of the polygon ($0 \leq i < m$), which stores all the n coordinates. In order words, `_point[i][j]` is jth coordinate of the ith point, where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$.

You are given the polygon class defined below:

```
class Polygon {
public:
  // default constructor
  // m is the number of points
  // n is the dimension of the points
  Polygon( unsigned int m = 0, unsigned int n = 0 ){
    cout << "default constructor" << endl;
    // codes no need to know or to implement
    // .....
  }
  Polygon( const Polygon& mt ); //copy constructor
  ~Polygon();           //destructor
  double* FindCentroid() const;  // return the centroid of the polygon

  const int getNumOfPoints() const { return _m; }
  const int getDimension() const { return _n; }

  //Accessor: Get the value stored at the m-th point and n-th dimension
  const double getValue( unsigned int m, unsigned int n ) const {
    if (m<0 || m>=_m || n<0 || n>=_n) {
      cerr << "ERROR: Index out of range" << endl;
      exit(-1);
    }
    else  return _point[m][n];
  }

  //Mutator: Set the value at the m-th point and n-th coordinate
  bool setValue( double & value, unsigned int m, unsigned int n ) {
    if (m>=0 && m<_m && n>=0 && n<_n){
      _point[m][n] = value;
      return true;
    }
    else
      return false;
  }

private:
  unsigned int _m;  // the number of points
  unsigned int _n;  // the dimension of every point
  double** _point;  // point[i]: the i-th point
                    //point[i][j]: the j-th coordinate of i-th point
};
```

(a) *(7 points)* Implement the copy constructor. Note that the input parameter `mt` may be a polygon which has no points (i.e., its `_point` is `NULL` and `_m = _n = 0`.)

```
// copy constructor
Polygon::Polygon( const Polygon& mt) {
    cout << "copy constructor" << endl;
    //implement your code below
```

(b) *(3 points)* Implement the destructor.

```
// destructor
Polygon::~Polygon() {
    cout << "destructor" << endl;
    //implement your code below
```

(c) *(8 points)* Implement the member function `FindCentroid()`, which returns the centroid of the polygon. Given a polygon of m points labeled as $x_0, x_1, x_2, \ldots, x_{m-1}$ and the coordinates of point i given by $(x_{i,0}, x_{i,1}, \ldots, x_{i,n-1})$, then the coordinate of the centroid is given by $(c_0, c_1, \ldots, c_{n-1})$ where

$$c_j = \frac{1}{m} \sum_{i=0}^{m-1} x_{i,j},$$

for $0 \leq j \leq n-1$.

`FindCentroid` returns a pointer to an array which holds the centroid of the polygon.

```
// Returns the centroid of the polygon as an array
double* Polygon::FindCentroid() const{
    //implement your code below
```

(d) *(12 points)* Write down the output of the following function:

```
void PrintPoints( Polygon p ){
  for( unsigned int i=0; i<p.getNumOfPoints(); i++ ){
    for( unsigned int j=0; j<p.getDimension(); j++ )
      cout << p.getValue(i,j) << ' ';
    cout << endl;
  }
}

int main() {
  Polygon p(3,4);
  double k = 0;
  for( int i=0; i<p.getNumOfPoints(); i++ )
    for( int j=0; j<p.getDimension(); j++ ){
      k += 1;
      p.setValue(k, i,j);
  }
  PrintPoints(p);
  double* centroid = p.FindCentroid();
  for( int n=0; n<p.getDimension(); n++ )
    cout << centroid[n] << " ";
  cout << endl;
  return 0;
}
```

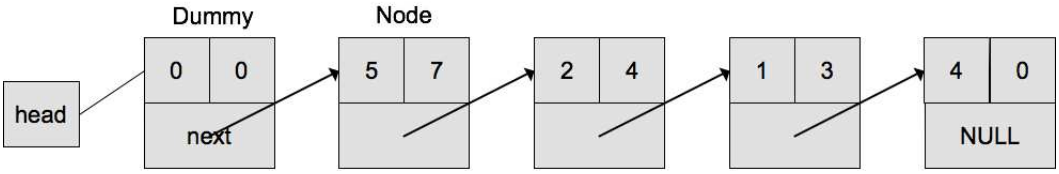
3. **Linked List: Polynomial** *(25 points total)*

A polynomial $f(x)$ of degree n is written as

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_i x^i + \ldots + c_1 x^1 + c_0 x^0, \quad (1)$$

where i is called the exponent of x^i, c_i is the coefficient of x^i, and $c_i x^i$ is called the ith term, $0 \leq i \leq n$.

In this problem, you will implement a polynomial ADT using a linked list with a dummy header (or sentinel) node. For example, we represent the polynomial $f(x) = 5x^7 + 2x^4 + 1x^3 + 4x^0$ by the following linked list, where the first node is a dummy whose term field has no meaning:



In the implementation, each term is stored as a `Term` object consisting of the coefficient and exponent. Each node in the linked list consists of a `Term` object and a `next` pointer pointing to the next node. The definitions of all the classes are shown below:

```
#define NodePointer Node*
typedef int CoefType;

// term in a polynomial
class Term{
public:
  CoefType coef;
  int expo;
};

// node in a linked list
class Node{
private:
  Term data;
  NodePointer next;
public:
  // node constructor
  Node (CoefType co = 0, int ex = 0, Node * ptr = NULL)  {
    data.coef = co;
    data.expo = ex;
    next = ptr;
  }
  friend class Polynomial;
};
```

The polynomial ADT with some of its supporting functions is given below:

```
// polynomial ADT
class Polynomial {
private:
  NodePointer head; // pointing to the first dummy node
public:
  // constructor
  Polynomial(CoefType* c = NULL, int* e = NULL, int num = 0);
  // destructor
  ~Polynomial();
  // add a term into the polynomial
  void add(CoefType c, int e);
  // differentiation
  void differentiate();
};
```

(a) *(8 points)* Implement the `add` member function in the polynomial class, which adds a term with coefficient `c` and exponent `e` to a polynomial object. Note that the term should be added into the polynomial object in such a way that the exponents are in *decreasing* order. If the exponent has already existed, their coefficients will be *added*.

```
// adding a term to the polynomial with decreasing exponents
// c: the coefficient
// e: the exponent
void Polynomial::add(CoefType c, int e){
```

(b) *(5 points)* Given the above, implement the constructor, which takes in an array of coefficients and their corresponding exponents to form a polynomial with decreasing exponents. Note that the exponents that `e` points to may not be sorted and may be repeated. If the exponents are repeated, their coefficients should be added together.

```
// constructor
// c: pointer to an array of coefficients
// e: pointer to the corresponding exponents
// num: the number of elements in c (or e)
// num >= 0
Polynomial::Polynomial(CoefType* c, int* e, int num){
```

(c) *(5 points)* Implement the destructor:

```
// destructor
Polynomial::~Polynomial(){
```

(d) *(7 points)* Implement the differentiation (or derivative) function which self-differentiates the polynomial. The derivative of $f(x)$ given by Equation (1) is defined as

$$f'(x) = nc_n x^{n-1} + (n-1)c_{n-1}x^{n-2} + \ldots + 2c_2 x + c_1 x^0. \tag{2}$$

```
void Polynomial::differentiate(){
```

4. **Overloading: BigInt** *(25 points total)*

In our tutorial , we have introduced the BigInt class, which can represent integers much larger than the maximum bound for `integer` type in computers.

You will deal with only *positive* BigInt in this problem. The BigInt class uses a dynamic array of `char` to represent the "big" integers. Each of the elements in the array can only be one of `'0'`, `'1'`, `'2'`, . . ., `'9'`, i.e., each array element stores just *one* digit of the "big" integer.

The BigInt.h file define the BigInt class as follows:

```
class BigInt{
 public:
  // constructor
  BigInt(){
    num = NULL;
    size = 0;
  };

  // convert an array of integral digits by tmp to BigInt
  // tmp: pointer to the array
  // length: the number of digits in the array
  BigInt(const int* tmp, int length);

  BigInt( const BigInt & );  // copy constructor: No need to implement

  // assignment: No need to implement
  const BigInt & operator=( const BigInt & );

  // destructor
  ~BigInt(){
    if (num !=NULL)
      delete[] num;
  };

  char & operator[] const (int index);

  int length() const {return size;};

 private:
  char* num;  // the big integer in char
  int size;   // number of digits in the big integer
  char* getNum(){return num;};
};
```

(a) *(2 points)* Implement the overloading of the operator `[]`, so that it returns the digit in `char` of a particular index. For example, if A is a BigInt object, we can get the 5th digit of A from the left in `char` by calling `A[4]` (digits are indexed $0, 1, 2, \ldots$ from the left).

```
char & BigInt::operator[] (int index){
 assert(index >= 0 &&  index < size);
// your code below
```

(b) *(5 points)* Implement the constructor which takes in an integer array of digits and converts it to a BigInt object. You may assume that the digits in the array are all integers between 0 and 9.

Hint: in order to convert a digit to a `char` type, you can add `('1'-1)` to it. For example, if you want to convert the digit 3 to a `char` in the form of `'3'`, you can use `3+('1'-1)`.

```
// convert an integer array of digits (0-9) to BigInt
// tmp: pointer to the array
// length: the number of digits in the integer array
BigInt(const int* tmp, int length){
```

(c) *(14 points)* In addition to the above, you need to implement prefix increment of BigInt, i.e., you want to support `++bi` for a BigInt `bi`. Write below the prototype of the member function in the BigInt class, and its implementation outside the class.

(d) *(4 points)* You also want to implement postfix increment which supports `bi++` for a BigInt `bi`. Write below the prototype of the member function and its implementation outside the class.