COMP 152
Fall 2010

Midterm Solutions

1. (a)

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

  if (fin.eof())
    return 0;

  char input;
  RingNode * tmp = current;

  fin.get(input);
  if (input == '\n')
    return 0;

  current = new RingNode;
  current->value = input;
  if( tmp == NULL )  // first node in the ring
    current -> next = current; // self-loop for a single node
  else  // other nodes in the ring
    current -> next = tmp;  // point back to the head to form a loop

  return 1+read_recursively(fin, current->next);
}
```

Using static variable, another solution is

```
//Alternative Solution
int read_recursively (ifstream& fin, RingNode*& current){
  static RingNode* head = NULL;

  current = head;
  if (fin.eof())
    return 0;

  char input;
  fin.get(input);
  if (input == '\n')
    return 0;

  current = new RingNode;
  current->value = input;
  if (head == NULL)
    head = current;

  return (1+read_recursively(fin, current->next));
}
```

(b) Note that the following recursive function deletes from the head. One can implement a recursive function which deletes from the tail (not shown).

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

  if (current == NULL)
    return;
  if (current != current->next){  // not a single node yet
    RingNode* tmp = current->next;
    current->next = tmp->next;
    delete tmp;
    delete_recursively (current);
  }
  else{ // a single node left
    delete current;
    current = NULL;
  }
}
```

2. (a)
```
Polygon::Polygon( const Polygon& mt){
  cout << "copy constructor" << endl;
  if( mt._point == NULL ){
    _m=0;
    _n=0;
    _point=NULL;
  }
  else{
    _m = mt._m;
    _n = mt._n;
    _point = new double*[_m];
    for( unsigned int i=0; i<_m; i++ ){
      _point[i] = new double[_n];
      for( unsigned int j=0; j<_n; j++ )
       _point[i][j] = mt._point[i][j];
    }
  }
}
```

(b)
```
Polygon::~Polygon(){
  cout << "destructor" << endl;
  if(_point!=NULL) {
    for( unsigned int i=0; i<_m; i++ )
      delete [] _point[i];
    delete [] _point;
  }
}
```

(c)
```
double* Polygon::FindCentroid() const{
  double* centroid = new double[_n];
  for( int j=0; j<_n; j++ ){
    centroid[j] = 0;
    for( int i=0; i<_m; i++ ) {
      centroid[j]+=_point[i][j];
    }
    centroid[j]/=_m;
  }
  return centroid;
}
```

(d)
```
default constructor
copy constructor
1 2 3 4
5 6 7 8
9 10 11 12
destructor
5 6 7 8
destructor
```

3. (a)
```
void Polynomial::add(CoefType c, int e){
  NodePointer terms = head;
  NodePointer t;
  while (terms->next!=NULL){
    if (terms->next->data.expo < e){ // find an internal position
      NodePointer temp = terms->next;
      t = new Node(c,e);
      terms->next = t;
      t->next = temp;
      return;
    }
    else if (terms->next->data.expo == e){ //clash with existing exponent
      terms->next->data.coef += c;
      return;
    }
    else terms = terms->next;
  }
  t = new Node(c,e);  // append to the tail
  terms->next = t;
}
```

(b)
```
Polynomial::Polynomial(CoefType* c, int* e, int num){
  NodePointer terms = new Node();
  head = terms;
  for (int i = 0 ; i< num ; i++) {
    add(c[i], e[i]);
  }
}
```

(c)
```
Polynomial::~Polynomial(){
  NodePointer terms = head ;
  while (terms!=NULL) {
    terms = terms->next;
    delete head;
    head = terms;
  }
}
```

(d)
```
void Polynomial::differentiate(){
  NodePointer terms = head;
  while (terms->next != NULL) {
    if (terms->next->data.expo == 0) { // constant term
      NodePointer temp = terms->next;
      terms->next = temp->next;
      delete temp;
    }
    else
      terms->next->data.coef =
terms->next->data.coef * terms->next->data.expo--;

    terms = terms->next;
  }
}
```

4. (a)
```
char& BigInt::operator[] const (int index){
  assert(index >= 0 &&  index < size);
  return num[index];
}
```

(b)
```
 BigInt(int* tmp, int length){
  num = new char[length];
  for (int i = 0; i < length;i++){
//    assert (tmp[i] >= 0 && tmp[i] <= 9);
    num[i] = tmp[i]+('1'-1); //Convert the digit to a char
  }
  size = length;
};
```

(c) `BigInt& operator++ ();`

```
BigInt& BigInt:: operator++(){

  int len = length();
  if (len == 0){ //The BigInt hasn't store any integer,just return '1'
    size++;  // size = 1;
    num = new char[1];
    num[0] = '1';
    return *this;
  }

  int carry = 1;// initialize carry as 1 for
                // the incrementation of the lowest bit
  int bits = len;//the length of the increment result
  char* result = new char[bits]; // the increment result

  len--;   // index to do digit-by-digit arithmetic
  int sum; // result of the sum of the digit and carry
  while (len>=0){
    sum = num[len]-('1'-1) + carry; // Add each digit to carry
    int tmp = sum % 10;
    carry = sum / 10;
    result[len] = tmp+('1'-1);  // convert to char
    len--;
  }

  if ( carry == 1){ // The length is increased by 1
    char* p = result;
    bits++;
    result = new char[bits];//enlarge the length in result
    result[0] = '1';
    for (int j=0;j<bits-1;j++) //copy the original "result" pointed by p
                               //to the new "result"
      result[j+1] = p[j];
    delete[] p;
    size++;
  }

  delete[] num;  // let num point to the new "result"
```

```
  num = result;
  return *this;
}
```

(d)
```
BigInt BigInt::operator ++(int);

BigInt BigInt::operator ++(int){
  BigInt result = *this;
  operator++();
  return result;
}
```