Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

CSI2372A
Advanced Programming Concepts with C++

MIDTERM EXAMINATION

Length of Examination: 75 minutes **November 9, 2016, 14:30**

Professor: Jochen Lang **Page 1 of** 14

Family Name: ____

Other Names: ____

Student Number: ___________

Signature________________________

You are allowed **ONE TEXTBOOK** as a reference. No calculators or other electronic devices are allowed.

Please answer the questions in this booklet. If you do not understand a question, clearly state an assumption and proceed.

At the end of the exam, when time is up: Stop working and turn your exam upside down. Remain silent.

Question	Marks	Maximum
A.1-A.3		3
B.1		3
B.2		3
B.3		3
C.1		1
C.2		1
C.3		2
C.4		3
C.5		3
C.6		4
Total		26

PART A: SHORT QUESTIONS (3 MARKS)

1. Change the following class to make it abstract

```
class A {
   int d_A;
 public:
   virtual void set( int a) = 0;
}

void A::set( int a ) {
  d_A = a;
}
```

2. What is printed by the following?

```
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {};
};

class D1 : public Base {
public:
  D1() = default;
};

class D2 : public Base {
};


int main() {
  D1 dA;
  Base* bA = &dA;
  Base& bB = dA;

  try {
        D2* d = dynamic_cast<D2*>(bA);
  } catch(...) {
        cout << "Error: bA" << endl;
  }
  try {
        D2& d = dynamic_cast<D2&>(bB);
  } catch(...) {
        cout << "Error: bB" << endl;
  }
}
```

Error: bB

3. The following class definition does not compile. Correct the error(s).

```
class Toto {
    const int d_data;
public:
    ~~Toto() { d_data = 20; }~~
    Toto() : d_data{20} { }
};
```

PART B: Short Programs (9 MARKS)

1. What is printed by the following program? [3]

```
#include <iostream>
using namespace std;

class Base {
   int d_b = 1;
public:
   Base() = default;
   Base( int b ) : d_b{b} {}
   int get() { return d_b; }
   virtual void set( int b) { d_b = b; }
   virtual void print() { cout << d_b << " "; }
};

class Derived : public Base {
   int d_d = 2;
public:
   Derived() = default;
   Derived( int d ) : d_d{d} {}
   virtual int get() { return d_d; }
   void set( int d) override { d_d = d; }
   virtual void print() { cout << d_d << " ";  }
};

int main() {
   Derived da(4), db, dc(3);
   da.print(); db.print(); dc.print(); cout << endl;
   Base* bPtr = &da;
   Base& bRef = db;
   Base bVal = dc;
   bPtr->print(); bRef.print(); bVal.print(); cout << endl;
   bPtr->set(5); bRef.set(6); bVal.set(7);
   cout << bPtr->get() << " " << bRef.get() << " " <<
           bVal.get() << endl;
   return 0;
}
```

```
4 2 3
4 2 1
1 1 7
```

2. Implement a deep assignment operator for the class `DArray`. [3]

```
class DArray {
  double* d_array;
  int d_size;
public:
  DArray(int sz) : d_size{sz} {
    d_array = new double[d_size];
  }
  ~DArray() {
    delete[] d_array;
  }
};
```

```
DArray& operator=(const DArray& oA) {
     if ( this == &oA ) return *this;
     delete d_array;
     d_array = new double[oA.d_size];
     d_size = oA.d_size;
     for ( int i=0; i<oA.d_size; ++i )
     {
      d_array[i] = oA.d_array[i];
     }
     return *this;
}
```

3. What is printed by the following program? [3]

```
#include <iostream>
using namespace std;

class Point {
  int d_x=1, d_y=0;
public:
  Point() = default;

  Point(int abs, int ord=0) : d_x{abs}, d_y{ord} {
    cout << "ctor: " << d_x << " " << d_y << "\n"; }

  Point(const Point &);

  Point& add( const Point& oP ) {
    d_x += oP.d_x; d_y += oP.d_y;
    return *this; }

  ~Point();
};

Point::Point(const Point& oP) : d_x{oP.d_x}, d_y{oP.d_y} {
  cout << "copy-ctor: " << d_x << " " << d_y << "\n"; }

Point::~Point () {
  cout << "dtor : " << d_x << " " << d_y << "\n"; }

void fct (Point d, Point * add) {
  cout << "start (fct)\n";
  delete add;
  cout << "end (fct)\n" ;
}

main () {
  cout << "start (main)\n" ;
  Point a, b = 2;
  Point c = a;
  Point* adr = new Point(3,3);
  fct (a, adr);
  cout << "end (main)\n";
}
```

```
start (main)
ctor: 2 0
copy-ctor: 1 0
ctor: 3 3
copy-ctor: 1 0
start (fct)
dtor : 3 3
end (fct)
dtor : 1 0
end (main)
dtor : 1 0
dtor : 2 0
dtor : 1 0
```

PART C: PROGRAMMING QUESTIONS (14 MARKS)

The class `LinkedList` holds a singly linked list of integers. Each integer is stored in an object of type Node with a field containing a number and a field containing a pointer to the following node. The `LinkedList` class is to use **internal aggregation** and hence it overloads the copy constructor, assignment operator and destructor. Consider the following definitions of the class `LinkedList` with its helper structure `Node`.

```
struct Node {
  int d_value ; // value of an element
  Node *d_next ; // pointer to the next node in the list
};

class LinkedList {
  Node *d_start; // pointer to the beginning of the list or null
  int d_nbElem; // the current number of elements - convenience
public:
  LinkedList(); // constructor creating an empty LinkedList
  LinkedList(const LinkedList&); // copy constructor
  ~LinkedList(); // destructor
  LinkedList& operator=(const LinkedList&);// assignment operator
  void add(int); // add an element to the list
  bool contains(int) const; // check if an element is in the list
  int nbElem() const;  // return number of elements in the list
};
```

1. Implement the default constructor `LinkedList()` to simply initialize a new `LinkedList` which is empty. [1]

```
LinkedList::LinkedList():
  d_start{nullptr},d_nbElem{0} {
}
```

2. Implement the accessor `nbElem()` to simply return the current number of elements in the list.[1]

```
int LinkedList::nbElem() const {
  return d_nbElem;
}
```

3. Implement `contains(int)` to return true if the integer `value` is in the list, false otherwise [2].

```
bool LinkedList::contains(int value) const {
  // loop over the linked list
  Node* eNode = d_start;
  while ( eNode != nullptr ) {
     if ( eNode->d_value == value )
          return true;
     eNode = eNode->d_next;
  }
  return false;
}
```

4. Implement `add(int)` to create a new Node and add an element to the linked list. [3]

```
#if 1
// adding at the end
void LinkedList::add(int _value) {
 // loop to the end - should keep a pointer
 // to the end
 Node** eNode = &d_start;
 while ( *eNode != nullptr )
     eNode = &((*eNode)->d_next);
 // new Node
 *eNode = new Node();
 (*eNode)->d_value = _value;
 (*eNode)->d_next = nullptr; // not needed
 ++d_nbElem;
 return;
}
#else
// adding at the front
void LinkedList::add(int _value) {
 // new Node
 Node* eNode = new Node();
 eNode->d_value = _value;
 eNode->d_next = d_start;
 d_start = eNode;
 ++d_nbElem;
 return;
}
#endif
```

5. Implement the destructor `~LinkedList()` You can assume that all `Node` objects have been dynamically allocated on the heap. `[3]`

```
LinkedList::~LinkedList() {
    while (d_start != nullptr) {
          Node* prev = d_start;
          d_start = d_start->d_next;
          delete prev;
    }
 }
```

6. Implement the copy constructor `LinkedList(const LinkedList&)` You must use internal aggregation. [4]

```
LinkedList::LinkedList(const LinkedList& oL) :
  d_start{nullptr},d_nbElem{oL.d_nbElem} {
    if ( oL.d_start != nullptr ) {
      // pointer to loop over source list
      Node* sNode = oL.d_start;
      // Create first node
      d_start = new Node( *sNode );
      // not needed because default C-Ctor
      d_start->d_next = nullptr;
      // pointer to the destination list
      Node* dNode = d_start;
      sNode = sNode->d_next; // source node
      while ( sNode != nullptr ) {
        dNode->d_next = new Node( *sNode );
        // advance both list
        dNode = dNode->d_next;
        dNode->d_next = nullptr;
        sNode = sNode->d_next;
      }
    }
}
```