Université d'Ottawa
Faculté de génie

School of Electrical
Engineering and
Computer Science

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

École de science
informatique et de
génie électrique

CSI2372 Advanced Programming Concepts with C++

FINAL EXAM

Length of Examination: 3 hours **December 19, 2015, 14:00**

Professor: Jochen Lang **Page 1 of 17**

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 close your exam booklet. Remain silent.

Question	Marks	Out of
1.1		4
1.2		4
1.3		4
2.1		3
2.2		3
2.3		3
3		7
4		10
Total		38

Section 1: Programming Questions (12 marks)

1. Consider the class Animal below and implement the function save which save a vector of animals to file by appending it to the file if the file exists. The function must return the number of animals written to file. You must consider write errors in the return value. [4]

```
class Animal {
  string d_name;
  int d_nLegs;

public:
  Animal(string name, int nLegs ) : d_name(name), d_nLegs(nLegs) {};

  const string toString() const {
    ostringstream os;
    os << d_name << " " << d_nLegs;
    return os.str();
  }
};


int save( const vector<Animal>& aVec, const string& fN = string("animals.txt"))
{
```

2. Sort the vector of building codes below with `std::sort` by `d_number` and print the vector. Next sort the vector again, this time sort by `d_code` in reverse order. Note that you are not allowed to modify the `struct BuildingCode` but that you have to implement a solution external to the `struct`.[4]

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last ); (1)

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp ); (2)
```

Sorts the elements in the range `[first, last)` in ascending order. The order of equal elements is not guaranteed to be preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function object `comp`.

Source: cppreference.com

```
struct BuildingCode {
  string d_code;
  string d_name;
  int d_num;

  friend ostream& operator<<( ostream& os, const BuildingCode& bc ) {
    os << bc.d_code << ": " << bc.d_name << " " << bc.d_num;
    return os;
  }
};

int main() {
  vector<BuildingCode> bcVec{{"STE","SITE",800},
                             {"MNT","Montpetit",125},
                             {"MRN","Marion",140},
                             {"VNR","Vanier",11},
                             {"LMX","Lamoureux",145},
                             {"MCD","MacDonald",150}};
```

(blank page)

3. Derive a class `FourLegged` from `Animal` that will always initialize the number `int Animal::d_nLegs` to 4. Make sure that your class works with the following main program. If necessary make any changes to the parent class `Animal` below. [4]

```
class Animal {
  string d_name;
  int d_nLegs;

public:
  Animal(string name, int nLegs ) : d_name(name), d_nLegs(nLegs) {};

  const string toString() const {
    ostringstream os;
    os << d_name << " " << d_nLegs;
    return os.str();
  }
};

ostream& operator<<( ostream& os, const Animal& animal ) {
  os << animal.toString();
  return os;
};

class FourLegged : public Animal {
   // Omitted – please implement on next page
};

int main() {
  FourLegged cheeta( "cheeta" );
  FourLegged lizard( "lizard", false );
  Animal bird( "duck", 2);

  Animal* animals[]{&cheeta,&lizard,&bird};

  for ( auto animalPtr:animals ) {
    cout << *animalPtr << endl;
  return 0;
}
```

Prints

```
cheeta 4 mammal
lizard 4 not a mammal
duck 2
```

Your implementation:

```
class FourLegged : public Animal {
```

Section 2: Abstract Data Types (9 MARKS)

Consider the class `Node` which holds a node in a four-connected graph (up to four neighbours per node).

```
 class Node {
   int d_element;
 protected:
   Node* d_neighbour[4]{0,0,0,0};

 public:
   Node( int element ) : d_element(element) {}

   Node( const Node& on ) : d_element(on.d_element) {
     for ( int i=0; i<4; ++i ) {
       if ( on.d_neighbour[i] )
          d_neighbour[i] = new Node(on.d_neighbour[i]->d_element);
     }
   }

   ~Node() {
     // Omitted – please implement on next page
   }

   bool add( Node& on ) {
     if ( d_neighbour[3] != 0 ) return false;
     d_neighbour[3] = new Node(on);
     rotate();
     return true;
   }

   void rotate() {
     Node* tmp(d_neighbour[0]);
     cout << "rot" << endl;
     for ( int i=0; i<3; ++i ) {
       d_neighbour[i] = d_neighbour[i+1];
     }
     d_neighbour[3] = tmp;
     cout << "stop" << endl;
   }


   friend ostream& operator<<( ostream& os, const Node& node ) {
     os << node.d_element << " ";
     for ( auto nghbPtr:node.d_neighbour ) {
       if ( nghbPtr ) os << *nghbPtr;
     }
     return os;
   }
 };
```

1. Implement the destructor for class Node. Carefully consider the constructors and make sure your destructor confirms with the class as implemented [3].

2. Implement the assignment operator for the class Node (operator=). Carefully consider the copy constructor and ensure that the assignment operator is consistent with the class [3].

3. Define a global compound assignment operator (not an in-class operator) which adds a new neighbour to the node. [3]
   This means that

```
if ((node1 += node2))  return "success";
```

   should behave identical to

```
if (node1.add(node2))  return "success";
```

PART 3: Constructors and Pointers (7 MARKS)

What will the following program print. Please find the questions on the following page and answer directly or on the extra page.

```
#include <iostream>
#include <vector>
#include <memory>

using std::cout;
using std::endl;

class Pair {

  public:
    int p1, p2;

    Pair() : p1(0), p2(0) {
        cout << "Empty" << endl;
    }

    Pair(int a, int b=0) : p1(a), p2(b) {
        cout << "Full" << endl;
    }

    Pair(const Pair& p) {
        p1= p.p1; p2= p.p2;
        cout << "Copy" << endl;
    }

    Pair& operator=(const Pair& p) {
        if (this != &p) {
            p1= p.p1; p2= p.p2;
        }
        cout << "Assignment" << endl;;
        return *this;
    }
};

void fct1(Pair& p) {
  p.p1 =0;
}

Pair fct2() {
  return Pair(2);
}

void fct3(Pair p) {
  p.p1= 0;
}
```

```
int main()
{
    cout << "Question 1.1" << endl;
    Pair pairs1[3];

    cout << "Question 1.2" << endl;
    Pair p(1,2);

    cout << "Question 1.3" << endl;
    Pair pairs2{(1,2),(3,4)};

    cout << "Question 1.4" << endl;
    std::vector<Pair> v(4,4);

    cout << "Question 1.5" << endl;
    p= v[0];

    cout << "Question 1.6" << endl;
    v[1]=p;

    cout << "Question 1.7" << endl;
    v.pop_back(); v.pop_back(); v.pop_back();

    cout << "Question 1.8" << endl;
    v.push_back(pairs1[1]);

    cout << "Question 1.9" << endl;
    std::shared_ptr<Pair> ptr1(new Pair(3));

    cout << "Question 1.10" << endl;
    std::shared_ptr<Pair> ptr2(ptr1);

    cout << "Question 1.11" << endl;
    fct1(p);

    cout << "Question 1.12" << endl;
    p= fct2();

    cout << "Question 1.13" << endl;
    fct3(p);

    cout << "Question 1.14" << endl;
    Pair *ptr3= &p;

    return 0;
}
```

(blank page)

Section 4: Containers of the Standard Template Library (10 marks)

The following class contains a set of elements. Each element is associated with a count which represents the number of times the element has been inserted in the class. Hence, when an element is inserted the first time, the value is set to 1. Each time, the method `add` is called with the same element the count is simply incremented by 1.

A `main` function is given at the end of this question to help clarify the intended functionality of the class.

Complete the code below:

```
template <typename T>
class MultiSetWithCount {
    // map containing one instance and its associated count
    std::map<T,int> d_store;

  public:
    // returns how many times this object was inserted
    int getCount(const T& object) {

        auto it= d_store.find(object);



        if (__________________________________________)
            return 0;
        else



            return ________________________________;
    }

    // Adds an object
    // If object is new then count starts at 1
    // If object already exists then simply increment count
    void add(const T& object) {
        d_store.insert(make_pair(object, 0)).first->second++;
    }
    // Remove an object. The object will be completely removed from the map
    // Returns the number of objects removed
    int remove(const T& object) {


        _____________________________ it= d_store.find(object);

        int numberOfDeletedObjects= 0;

        if (it!= d_store.end()) {


            ___________________________________________;



            ___________________________________________;
        }

        return numberOfDeletedObjects;
    }

    // Returns a vector containing the objects
    // For each object the number of copies to create for the vector
    // corresponds to the count value
    std::vector<T> toVector() {

        std::vector<T> v;


        for (auto iterator= ________________________________________) {



            int count= ______________________________________________;

            for (int i=0; i< count; i++) {



                v.push_back(________________________________________);
            }
        }

        return v;
    }
    // prints all values and counts
    void print() {



        for (auto iterator=________________________________________) {




            std::cout <<____________________________________________;
        }
        std::cout << std::endl;
    }
};

int main()
{
    MultiSetWithCount<std::string> ms;

    ms.add(std::string("red"));
    ms.add(std::string("red"));
    ms.add(std::string("green"));
    ms.add(std::string("green"));
    ms.add(std::string("blue"));
    ms.add(std::string("green"));

    std::cout << "green count is " <<
                  ms.getCount(std::string("green")) << std::endl;
    std::cout << ms.remove(std::string("red")); << std::endl;
    ms.print();

    std::vector<std::string> v= ms.toVector();
    std::cout << "vector size is " << v.size() << std::endl;

    return 0;
}

/* Console output:

green count is 3
2
(blue,1)(green,3)
vector size is 4

*/
```

std::map::find

```
iterator find( const Key& key );
```

Finds an element with key equivalent to `key`.

Parameters

key	-	key value of the element to search for

Return value

Iterator to an element with key equivalent to `key`. If no such element is found, past-the-end (see end()) iterator is returned.

std::map::erase

```
 void erase( iterator pos );                       (1)
size_type erase( const key_type& key );             (2)
void erase( iterator first, iterator last );        (3)
```

Removes specified elements from the container.

(1) Removes the element at `pos`.
(2) Removes the element (if one exists) with the key equivalent to `key`.
(3) Removes the elements in the range `[first; last)`, which must be a valid range in `*this`.

References and iterators to the erased elements are invalidated. Other references and iterators are not affected.

The iterator `pos` must be valid and dereferenceable. Thus the end() iterator (which is valid, but is not dereferencable) cannot be used as a value for `pos`.

Parameters

pos	-	iterator to the element to remove
first, last	-	range of elements to remove
key	-	key value of the elements to remove

Return value

(1) (3) Iterator following the last removed element.
(2) Number of elements removed.

Source: cppreference.com