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 21, 2017, 9:30**

**Professor: Jochen Lang** 

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 |
|---|---|---|
| A.1 | | 2 |
| A.2 | | 1 |
| A.3 | | 1 |
| B | | 7.5 |
| C.1 | | 2 |
| C.2-3 | | 3 |
| C.4 | | 2 |
| C.5 | | 3 |
| C.6 | | 3 |
| D.1-D.3 | | 4.5 |
| D.4-D.5 | | 3.5 |
| D.6-D.7 | | 3 |
| D.8-D.9 | | 3.5 |
| Total | | 38 |

## *Part A: Short Questions (4 marks)*

1. Write a program that opens the text file `final.txt` and saves the string `CSI2372` in it. Make sure to close the stream. Print an error message to console if the file cannot be opened. [2]

```
#include <iostream>
#include <fstream>

int main() {









 return 0;
}
```

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

```
#include <iostream>
#include <sstream>
#include <string>

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

class Tree {
  float d_height{0.0f};
  int d_age{0};
public:
  Tree(float height, int age) : d_height(height), d_age(age) {}
  virtual float annualGrowth()=0;
  string toString() const {
    std::ostringstream oss;
    oss << d_age << " : " << d_height;
    return oss.str();
  }
protected:
  float& getHeight() { return d_height; }
  int& getAge() { return d_age; }
};

class Oak : public Tree {
public:
  using Tree::Tree;
  float annualGrowth() override {
    getHeight() += 0.6;
    getAge() += 1;
  }
};

std::ostream& operator<<( std::ostream& os, const Tree& t ) {
  os << t.toString() << endl;
  return os;
}


int main() {
  Oak a{ 10.0, 15 };
  Tree& t = a;
  t.annualGrowth();
  Oak& o = dynamic_cast<Oak&>(t);
  cout << o << endl;
  return 0;
}
```

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

```
#include <iostream>
#include <string>

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

class PageSize {
  string d_name{};
public:
  PageSize( string name ) : d_name{name} {}
  inline string getName() const { return d_name; }
};

class Letter : PageSize {
public:
  Letter() : PageSize("Letter") {};
  string getName() const { return PageSize::getName(); }
};

struct DIN : PageSize {
  DIN( const string& name ) : PageSize( "DIN "+name ) {}
};

class A4 : public DIN {
public:
  A4() : DIN( "A4" ) { }
};


int main() {
  Letter l;
  A4 p;
  cout << l.getName() << endl;
  cout << p.getName() << endl;
  return 0;
}
```

## *Part B: Abstract Data Types (7.5 MARKS)*

The following listing contains global and in class operators for the given class `TrafficCounter`. A Traffic.

A `main` function and the expected print out is given at the end of the declarations to help clarify the intended functionality of the operators and the class.

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

struct CarCounter { int d_count; };
struct BikeCounter {  int d_count; };

// Global operators
class TrafficCounter;
ostream& operator>>( ostream& os, const TrafficCounter& tc );
istream& operator>>( istream& is, const TrafficCounter& tc );
TrafficCounter operator+( const TrafficCounter& tcA,
                          const TrafficCounter& tcB );

class TrafficCounter {
  CarCounter d_nCars{0};
  BikeCounter d_nBikes{0};

public:
  TrafficCounter() = default;
  TrafficCounter( const BikeCounter& bc );
  TrafficCounter( const CarCounter& cc );

  inline TrafficCounter& operator+=(const TrafficCounter& oc );
  inline TrafficCounter& operator+=(const BikeCounter& bc );
  inline TrafficCounter& operator+=(const CarCounter& cc );

  inline TrafficCounter operator++( int );
  inline TrafficCounter& operator++( );

  inline operator BikeCounter();
  inline operator CarCounter();

  friend ostream& operator<<( ostream& os, const TrafficCounter& tc );
  friend istream& operator>>( istream& is, TrafficCounter& tc );
};
```

```
int main() {
   TrafficCounter tcA;
```

```
    BikeCounter bc{1};
    CarCounter cc{2};

    tcA += bc;
    tcA += cc;
    cout << tcA << endl;

    TrafficCounter tcB = bc + tcA;
    cout << tcB << endl;

    tcA += tcB;
    cout << tcA << endl;

    TrafficCounter tcC = tcA++;
    cout << tcC << endl;

    tcC = ++tcA;
    cout << tcC << endl;
    return 0;
}

/* Console Output */

2 1
2 2
4 3
4 3
6 5
```

Complete the code starting on the next page in the space provided. (Note if you need less lines that is perfect, needing more likely indicates a problem).

1. Construct a `TrafficCounter` from a `CarCounter` with 0 `d_nCars` [0.5]

```
TrafficCounter::TrafficCounter( const CarCounter& cc ) :
   _____________________________________________
   _____________________________________________
{}
```

2. Construct a `TrafficCounter` from a `BikeCounter` with 0 `d_nBikes` [0.5]

```
TrafficCounter::TrafficCounter( const BikeCounter& bc ) :
   _____________________________________________
   _____________________________________________
{}
```

3. Add the bike and car counts of the argument `oc` to this [0.5]

```
TrafficCounter& TrafficCounter::operator+=(const TrafficCounter& oc ) {
   _____________________________________________
   return ______________________________________;
}
```

4. Add the bike count of the argument `bc` to this [0.5]

```
TrafficCounter& TrafficCounter::operator+=(const BikeCounter& bc ) {
   _____________________________________________
   return ______________________________________;
}
```

5. Add the car count of the argument `cc` to this [0.5]

```
TrafficCounter& TrafficCounter::operator+=(const CarCounter& cc ) {
   _____________________________________________
   return ______________________________________;
}
```

6. Increment the counts `d_nBikes` and `d_nCars` in this [0.5]

```
TrafficCounter& TrafficCounter::operator++( ) {
   ____________________________________________

   ____________________________________________

   return ____________________________________________;
}
```

7. Increment the counts `d_nBikes` and `d_nCars` in this [0.5]

```
TrafficCounter TrafficCounter::operator++( int ) {

   ____________________________________________

   ____________________________________________

   return ____________________________________________;
}
```

8. Convert this to a `BikeCounter` ignoring `d_nCars` [0.5]

```
TrafficCounter::operator BikeCounter() {

   return ____________________________________________;
}
```

9. Convert this to a `CarCounter` ignoring `d_nBikes` [0.5]

```
T TrafficCounter::operator CarCounter() {

   return ____________________________________________;
}
```

10. Implement the global `operator+` adding two `TraficCounter` [1]

```
TrafficCounter operator+( const TrafficCounter& tcA,
                          const TrafficCounter& tcB ) {

   _____________________________________________

   _____________________________________________

   return _____________________________________________;
}
```

11. Implement the insertion operator for `TrafficCounter` [1]

```
ostream& operator<<( ostream& os, const TrafficCounter& tc ) {

   _____________________________________________

   return _____________________________________________;
}
```

12. Implement the extraction operator for `TrafficCounter` [1]

```
istream& operator>>( istream& is, TrafficCounter& tc ) {

   _____________________________________________

   if ( !is ) _____________________________________________

   return _____________________________________________;

}
```

## *PART C: Internal Aggregations, Recursions and Callables (13 MARKS)*

Consider the following n-ary tree template.

```
#include <vector>
#include <iostream>
#include <functional>

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

template <class T> struct Node {
  T value;
  std::vector<Node<T>*> children;
};

template <class T>
class Tree {
  Node<T>* d_root{nullptr};
public:
  Tree()=default;
  Tree( const Tree& oT );
  Tree<T>& operator=( const Tree<T>& oT);
  ~Tree();
  // add a child to the first node where isParent returns true
  bool add( std::function<bool (const Node<T>* )>(isParent),
            const T& childVal );
  void depthFirstTraversal( std::function<bool ( Node<T>* )>(visitor) );
protected:
  bool depthFirstTraversal( Node<T>* curr,
                            std::function<bool ( Node<T>* )>(visitor) );
private:
  // internal helper function to delete all nodes
  void deleteAll( Node<T>* curr);
  // internal helper function to copy all nodes
  void copyTraversal( const Node<T>* src, Node<T>*& dst);
};

// Private helper functions
template <class T>
void Tree<T>::copyTraversal( const Node<T>* src, Node<T>*& dst)  {
  dst->value = src->value;
  for ( auto&  n : src->children ) {
    Node<T>* child = new Node<T>();
    dst->children.push_back(child);
    copyTraversal( n, *(--dst->children.end()));
  }
  return;
}
```

```
template <class T>
void Tree<T>::deleteAll( Node<T>* curr ) {
  if ( curr == nullptr ) return;
  std::vector<Node<T>*> cchildren{curr->children};
  delete curr;
  for ( auto n : cchildren ) {
    deleteAll( n );
  }
  return;
}
```

1. Implement the copy constructor for the class `Tree<T>` using the helper functions copyTraversal and /or deleteAll as appropriate. Each class Tree<T> must hold its own copy of the nodes (internal aggregation) [2]

2. Implement the assignment operator for the class `Tree<T>` using the helper functions copyTraversal and /or deleteAll as appropriate. Each class Tree<T> must hold its own copy of the nodes (internal aggregation) [2]

3. Implement the destructor for the class `Tree<T>` using the helper functions copyTraversal and /or deleteAll as appropriate. [1]?

4. Complete the code below implementing depth first traversal visiting nodes with the given visitor function. You may want to have a look at the given implementation of copyTraversal and deleteAll for clarification [2]

```
template <class T>
void Tree<T>::depthFirstTraversal(
                std::function<bool ( Node<T>* )>(visitor) ) {
  if ( d_root == nullptr ) return;
  else Tree::depthFirstTraversal( d_root, visitor);
  return;
}

template <class T>
bool Tree<T>::depthFirstTraversal( Node<T>* curr,
                std::function<bool ( Node<T>* )>(visitor) ) {
  bool cnt = visitor( curr ); // visit and continue?
  if (!cnt) { return cnt; }
  for ( auto& n : curr->children ) {
    // visit all children in the subtrees depth first


    _____________________________________________


    _____________________________________________


    _____________________________________________


    _____________________________________________

  }
  return cnt;
}
```

5.

Complete the code below implementing the insertion operator using the function `Tree<T>::depthFristTraversal`. Supply the missing arguments to the function call [3].

```
template <class T>
std::ostream& operator<<( std::ostream& os, const Tree<T>& t) {
  Tree<T> tmp = const_cast<Tree<T>& >(t);
  tmp.depthFirstTraversal(

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    );
  os << endl;
  return os;
}
```

6. The function add inserts a new node based on the given inserter function. Complete the add function by supplying the arguments to `depthFirstTraversal` finding the place to insert the new node and inserting it at the found place. [3]

```
template <class T>
bool Tree<T>::add( std::function<bool (const Node<T>* )> isParent,
                   const T& childVal ) {
  if ( d_root == nullptr ) {
    if ( isParent(nullptr) ) { // isParent must handle null pointer
      d_root = new Node<T>();
      d_root->value = childVal;
      return true;
    } else {
      return false;
    }
  } else {
    return !depthFirstTraversal(

          ____________________________________________

          ____________________________________________

          ____________________________________________

          ____________________________________________

          ____________________________________________

          ____________________________________________

          ____________________________________________

          ____________________________________________

    );
  }
}
```

## *Part D: Containers of the Standard Template Library (13.5 marks)*

The following class aggregates a `std::map` to implement a job list with orders. You will need to use the standard template library algorithm `std::find` and/or `std::find_if` for your implementations of the functions `find`, and `std::copy and/or std::copy_if` for your implementations of the functions `all` and `ordered`.

```
#include <iostream>
#include <map>
#include <vector>
#include <list>
#include <functional>
#include <exception>
#include <string>
#include <algorithm>

using std::cout;
using std::endl;
using std::string;
using std::ostream;

struct Job {
  string d_description;
  float d_price;
  bool d_complete;
};

ostream& operator<<( ostream& os, const Job& j ) {
  os << j.d_description << " " << j.d_price;
  if ( j.d_complete )  os << " " << "complete";
  return os;
}

class NoSuchJob : public std::exception {
};

class JobList;
ostream& operator<<( ostream& os, const JobList& jl );
```

```
class JobList {
  std::map<int,Job> d_orders;
  static int lastId;
public:
  JobList()=default;
  // Add a job to the orders assigning an unique id, return the id
  int order(const Job& jb);
  // return a copy of all jobs in std::vector
  std::vector<std::pair<int,Job> > all();
  // return a copy of all completed job orders in a std::list
  std::list<std::pair<int,Job> > complete();
  // return the ordered job for a given id
  const Job& getJob(int id);
  // return the id for a given job order
  int getOrderId(const Job& jb);
  // find an order where the function equiv is true and return the id
  int find(std::function<bool (const Job&)> equiv);
  // return the lowest ID of any order
  int getNextOrder();
  // mark the job with the given id as complete
  void complete( int id );
  // remove the job with the given id
  Job remove( int id );
  friend ostream& operator<<( ostream& os, const JobList& jl );
protected:
  static int uniqueID();
};

int JobList::uniqueID() {
     return JobList::lastId++;
}

int JobList::lastId=0;
```

# std::find, std::find_if

Defined in header <algorithm>

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );                    (1)
template< class InputIt, class UnaryPredicate > InputIt find_if( InputIt first, InputIt last,
 UnaryPredicate p );                                                            (2)
```

Returns the first element in the range `[first, last)` that satisfies specific criteria:

1) `find` searches for an element equal to `value`
2) `find_if` searches for an element for which predicate `p` returns true

## Parameters

first, last - the range of elements to examine

value - value to compare the elements to

p - unary predicate which returns true for the required element. The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have const &, but the function must not modify the objects passed to it.
The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

- `InputIt` must meet the requirements of `InputIterator`.
- `UnaryPredicate` must meet the requirements of `Predicate`.

## Return value

Iterator to the first element satisfying the condition or `last` if no such element is found.

# std::copy, std::copy_if

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );                                   (1)

template< class InputIt, class OutputIt, class UnaryPredicate >OutputIt
copy_if( InputIt first, InputIt last, OutputIt d_first, UnaryPredicate pred );                    (2) (since C++11)
```

Copies the elements in the range, defined by `[first, last)`, to another range beginning at `d_first`.

1) Copies all elements in the range `[first, last)`. The behavior is undefined if `d_first` is within the range `[first, last)`. In this case, std::copy_backward may be used instead.
2) Only copies the elements for which the predicate `pred` returns true. The relative order of the elements that are copied is preserved. The behavior is undefined if the source and the destination ranges overlap.

## Parameters

| | |
|---|---|
| first, last | - the range of elements to copy |
| d_first | - the beginning of the destination range. |
| pred | - unary predicate which returns true for the required elements. The signature of the predicate function should be equivalent to the following:<br><br>bool pred(const Type &a);<br><br>The signature does not need to have const &, but the function must not modify the objects passed to it.<br>The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type. |

Type requirements
- InputIt must meet the requirements of InputIterator.
- OutputIt must meet the requirements of OutputIterator.
- ForwardIt1, ForwardIt2 must meet the requirements of ForwardIterator.
- UnaryPredicate must meet the requirements of Predicate.

## Return value

Output iterator to the element in the destination range, one past the last element copied.

1. Implement the order function that adds a new Job with an unique id to the orders and returns the id. Use the static function uniqueId in JobList [1]

```
int JobList::order(const Job& jb) {

    _____________________________________________

    _____________________________________________
}
```

2. Implement the all function that returns all the orders as a vector. Your implementation must use std::copy [1.5]

```
std::vector<std::pair<int,Job> > JobList::all() {

    _____________________________________________

    _____________________________________________

    _____________________________________________
}
```

3. Implement the complete function that returns the completed jobs as a list. Your implementation must use std::copy_if [2]

```
std::list<std::pair<int,Job> > JobList::complete() {

    _____________________________________________

    _____________________________________________

    _____________________________________________
}
```

4. Implement the `getJob` function that returns a job with the given id. If the id does not exist throw the `NoSuchJob` exception. [1.5].

```
const Job& JobList::getJob(int id) {
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
}
```

5. Implement the `getOrderId` function that returns the id of a job order. If the job order does not exist throw the `NoSuchJob` exception. [2].

```
int JobList::getOrderId(const Job& jb) {
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
    _____________________________________________
}
```

6. Implement the function `find` that finds the first entry in the job orders for which the passed function evaluates to true. If the job does not exist throw the `NoSuchJob` exception. Your implementation must use std::find_if [2].

```
int JobList::find(std::function<bool (const Job&)> equiv) {

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    _____________________________________________

    return _____________________________________________;
}
```

7. Implement the function `getNextOrder` that returns the id of the job which is next, i.e., which has the lowest unique id. If there is no job order, throw the `NoSuchJob` exception. [1].

```
int JobList::getNextOrder() {

    _____________________________________________

    return _____________________________________________;
}
```

8. Implement the `complete` function that marks a job with the given id as complete. If there is no ordered job with that id, throw the `NoSuchJob` exception. [1.5]

```
void JobList::complete( int id ) {

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________


   return _____________________________________________________;
}
```

9. Implement the `remove` function that returns and removes the job a job with the given id from the job orders. If there is no job order, throw the `NoSuchJob` exception. [2]

```
Job JobList::remove(int id) {

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________


   return _____________________________________________________;
}
```