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 12, 2018, 14:00**

Professor: Jochen Lang **Page 1 of 19**

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.

Question	Marks	Out of
A		6
B		6
C		15
D		4
E		7
Total		38

At the end of the exam, when time is up: Stop working and close your exam booklet. Remain silent.

Part A: Short Questions (6 marks)

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

```
class Bird {
public:
  virtual void name() { cout << "Bird" << endl; }
  void numLegs() { cout << 2 << endl; }
  bool fly() {return true;}
};

class Emu : public Bird {
public:
  void name() { cout << "Emu" << endl; }
  void numLegs() { cout << 3 << endl; }
  bool fly() {return false;}
};

int main() {
  Emu e;
  Bird& pb = e;
  Bird b;
  cout << "1: ";
  if (e.fly()) e.name();
  cout << "2: ";
  if (pb.fly()) pb.name();
  cout << "3: ";
  if (b.fly()) b.name();

  return 0;
}
```

1: 2: Emu
3: Bird

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

```
int array[][3]{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int (&ptr)[3] = array[1];
std::cout << *(ptr+1);
```

5

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

```
vector<int> v{1,2,3,4,5,6};
auto ret= std::find(v.begin(),v.end(),0);
std::cout << *(ret-1);
```

6

4. What is printed by the following statements? [1]

```
std::set<string> s{"red","green","yellow","blue"};
for (auto x : s) {
  std::cout << x << " ";
}
```

~~green red yellow~~

5. What is printed by following statements? [1]

```
std::vector<unique_ptr<Shape>> v;
std::unique_ptr<Shape> p= make_unique<Shape>(Shape("Triangle"));
v.push_back(std::move(p));
if (p!=nullptr) std::cout<<"my pointer";
if (v[0]!=nullptr) std::cout<<"my vector";
```

my vector

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

```
class Person {
 public:
    Person(int x)  { cout << "Person ctor called" << endl;   }
};

class Faculty : public Person {
 public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty ctor called"<< endl;
    }
};

class Student : public Person {
 public:
    Student(int x):Person(x) {
        cout<<"Student ctor called"<< endl;
    }
};

class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x)    {
        cout<<"TA ctor called"<< endl;
    }
};

int main()  {
    TA ta1(123);
}
```

Person ctor called
Faculty ctor called
Person ctor called
Student ctor called
TA ctor called

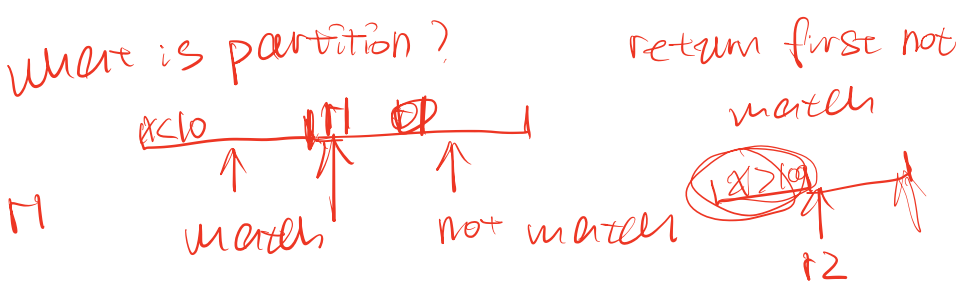
Part B: Complete the Code (6 MARKS)

1. Change the function division such that it throws a "Division by zero!" exception if the argument b equals 0. [1]

```
double division(int a, int b) {
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (string msg) {
      cerr << msg << endl;
   }

   return 0;
}
```

if (b == 0) {
throw("Division by zero");
} else
return.

```
double division(int a, int b) {
  if ( b == 0 ) throw string("Division by zero!");
  return (a/b);
}
```



2. Vector v1 contains a set of integers and the following two calls to `std::partition` are executed. [2]

```
auto r1= std::partition(v1.begin(),v1.end(),[](int x){ return x<10;});
auto r2= std::partition(r1,v1.end(),[](int x){ return x>100;});
```

Use the `std::copy` function to copy all the elements between 10 and 100 from vector v1 into the vector v2.

```
std::vector<int> v2;
```

std::copy(r2,v1.end(),std::back_inserter(v2));

add elements are the end!

3. Add a lambda function as the last parameter in the call to for_each so that 10 is added to each element in the vector less than 10. [1.5]

```
std::vector<int> v={2,15,22,3,32,5,7,44};

// expected result: {12,15,22,13,32,15,17,44}
for_each(v.begin(),v.end(),
```

____[](int& x) { if (x<10) x+=10; }_____);

4. Write a for loop to display the values of the vector v in reverse order, i.e. from last to first element. [1.5]

```
std::vector<int> v={2,15,22,3,32,5,7,44};
```

```
for ( auto iter = w.crbegin(); iter < w.crend(); ++iter )
 {
      cout << *iter << " ";
  }
```

Part C: Containers (15 MARKS)

The following class defines a container for elements. It adds the elements by copy.

```
template <typename T>
class Ring {
  // private internal node
  struct Node {
    T element;
    Node* next;
  };

  Node *current=nullptr;

public:

  // add an element after current
  void add(const T& element) {
    // create new element
    auto newNode= new Node({element,nullptr});
    // point to itself if single element
    newNode->next= newNode;
    // insert element in non-empty list
    if (current) {
      newNode->next= current->next;
      current->next= newNode;
    }
    // current becomes inserted element
    current= newNode;
  }

  // move current to next element and return it
  // return a default object if empty list
  T next() {
    if (current) {
      current= current->next;
      return current->element;
    } else {
      return T();
    }
  }

  Ring() = default;
};
```

1. Implement the copy constructor for the class `Ring` implementing a deep copy strategy. [3]

```
Ring( const Ring<T>& o) {
  if ( o.current ) {
    auto ptr = o.current;
    do {
       add(ptr->element);
       ptr = ptr->next;
    } while ( ptr != o.current );
    // go one notch further
    next();
  }
 }
```

2. Implement the destructor for the class `Ring` avoiding any memory leaks. [3]

```
~Ring() {
   if (current) {
     auto ptr = current;
     do {
        auto next = ptr->next;
        delete ptr;
        ptr = next;
     } while ( ptr != current );
   }
 }
```

3. Implement the assignment operator for the class `Ring` implementing a deep assignment strategy. [4]

```
Ring<T>& operator=( const Ring<T>& o) {
 if ( &o == this ) return *this;
 // delete own elements
 if (current) {
   auto ptr = current;
   do {
      auto next = ptr->next;
      delete ptr;
      ptr = next;
   } while ( ptr != current );
 }
 current = nullptr;
 if ( o.current ) {
   auto ptr = o.current;
   do {
      add(ptr->element);
      ptr = ptr->next;
   } while ( ptr != o.current );
   next();
 }
 return *this;
}
```

4. Implement the move constructor for the class `Ring`. [2]

```
Ring( Ring<T>&& rval ) noexcept :
  current(rval.current)
 {
  cout << "move ctor" << endl;
  rval.current = nullptr;
 }
```

5. Implement the move assignment operator for the class `Ring`. [3]

```
Ring<T>& operator=( Ring<T>&& rval ) noexcept
{
    if ( this != &rval ) { // could be replaced by assert
     // delete own elements
     if (current) {
        auto ptr = current;
        do {
          auto next = ptr->next;
          delete ptr;
          ptr = next;
        } while ( ptr != current );
     }
     current = rval.current;
     rval.current = nullptr;
   }
   return *this;
 }
```

Part D: Abstract Data Types (4 MARKS)

The Team class contains a list of players (`Player` class) contained in a `std::Map` in which the player's number serves as a key.

```
class Player;
class Team {

  std::Map<int,Player> players();

  …
```

a) Complete the definition of the method `addPlayer` to add a player to the team by specifying a number. If the specified number is already assigned to another player, this method returns false and does not insert. Otherwise, the player is added to the `std::Map` and the method returns true.

```
bool Team::addPlayer(int number, const Player& p) {

  auto ret= players.insert(std::make_pair(number,p));

  return ret.second;
}
```

access the value of current

b) Complete the definition of the `getNewNumber` method to obtain an unassigned player number from a starting number. The function must work with `const Team`.

```
int Team::getNewNumber(int startNumber) const {

  while (players.find(startNumber) != players.end())
    startNumber++;

  return startNumber;
}
```

c) Complete the definition of the `operator!` returning false if the team has fewer than 10 players.

```
bool Team::operator!() const {

  return players.size()< 10;
}
```

d) Complete the definition of the insertion operator (`operator <<`) allowing to display at the console the list of the numbers of players of the team.

```
ostream& operator<<(ostream &out,
                    const Team &team) {

  for( auto it = team.players.begin(); it != team.players.end(); ++it) {

    out << it->first << "\n";
  }

  return out;
}
```

Part E: Dynamic Allocation and Memory Management (7 marks)

Given the following definitions:

```
class Fruit {};

Fruit *produceFruit() {
  return new Fruit();
}

void consumeFruit(Fruit *ff) {
  delete ff;
}

Fruit* reproduceFruit(Fruit fruit) {
   Fruit* ret= new Fruit(fruit);
   return ret;
}
```

1. The function `leaky()` below includes memory leaks, however, it runs without other problems. Insert calls to delete in the function `leaky` such that no memory leak occurs. [3]

```
void leaky() {
  Fruit *f1= new Fruit();
  Fruit *f2= f1;
  Fruit *f3= new Fruit(*f1);

  delete f3;
  f3= reproduceFruit(*f1);



  consumeFruit(f3);
  f3= produceFruit();


  Fruit f4;
  f4= *f3;
  delete f1;
  delete f3;
  return;
}
```

2. Fix the original memory leaks by replacing **all** regular pointers with smart pointers. Use `std::unique_ptr` whenever possible, otherwise use `std::shared_ptr`. [4]

```
#include <memory>
using std::unique_ptr;
using std::shared_ptr;
using std::move;



unique_ptr<Fruit>  produceFruit() {

  return unique_ptr<Fruit>(new Fruit());
}



void consumeFruit( unique_ptr<Fruit> ff) {


}



unique_ptr<Fruit> reproduceFruit(Fruit fruit) {

   unique_ptr<Fruit> ret = unique_ptr<Fruit>(

            new Fruit(fruit));
   return ret;
}
```

```
void leaky() {
      shared_ptr<Fruit> f1
            = shared_ptr<Fruit>( new Fruit() );
      shared_ptr<Fruit> f2= f1;
      unique_ptr<Fruit> f3
            = unique_ptr<Fruit>( new Fruit(*f1));
      f3= reproduceFruit(*f1);

      consumeFruit( std::move(f3));
      f3= produceFruit();
      Fruit f4;
      f4= *f3;
      return;
}
```

std::copy

Defined in header <algorithm>

template< class InputIt, class OutputIt >
OutputIt copy(InputIt first, InputIt last, OutputIt d_first); (until C++20) (1)
template< class InputIt, class OutputIt >
constexpr OutputIt copy(InputIt first, InputIt last, OutputIt d_first); (since C++20)

… (omitted)

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)` starting from first and proceeding to last - 1. The behavior is undefined if `d_first` is within the range `[first, last)`. In this case, std::copy_backward may be used instead.

Parameters

first, last - the range of elements to copy
d_first - the beginning of the destination range.

Return value

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

std::find

Defined in header `<algorithm>`

template< class InputIt, class T >
InputIt find(InputIt first, InputIt last, const T& value); (1) (until C++20)
template< class InputIt, class T >
constexpr InputIt find(InputIt first, InputIt last, const T& value); (since C++20)

… (omitted)

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

1) `find` searches for an element equal to `value`

Parameters

first, last - the range of elements to examine
value - value to compare the elements to

Return value

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

std::for_each

Defined in header <algorithm>

template< class InputIt, class UnaryFunction >
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f); (until C++20)
(1)
template< class InputIt, class UnaryFunction >
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f); (since C++20)

… (omitted)

1) Applies the given function object `f` to the result of dereferencing every iterator in the range `[first, last)`, in order.

For both overloads, if the iterator type is mutable, `f` may modify the elements of the range through the dereferenced iterator. If `f` returns a result, the result is ignored.

Unlike the rest of the algorithms, for_each is not allowed to make copies of the elements in the sequence even if they are trivially copyable.

Parameters

first, last - the range to apply the function to

f - function object, to be applied to the result of dereferencing every iterator in the range `[first, last)`

The signature of the function should be equivalent to the following:

void fun(const Type &a);

The signature does not need to have const &.
The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Return value

1) `f` (until C++11) std::move(f) (since C++11)
2) (nothing)

std::partition

Defined in header <algorithm>

template< class BidirIt, class UnaryPredicate >
BidirIt partition(BidirIt first, BidirIt last, UnaryPredicate p); (until C++11)

(1)

template< class ForwardIt, class UnaryPredicate >
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p); (since C++11) (until C++20)

… (omitted)

1) Reorders the elements in the range `[first, last)` in such a way that all elements for which the predicate `p` returns true precede the elements for which predicate `p` returns false. Relative order of the elements is not preserved.

Parameters

first, last - the range of elements to reorder

Return value

Iterator to the first element of the second group.

Source: cppreference.com
License: Creative Commons Attribution-Sharealike 3.0 Unported License