Université d'Ottawa
Faculté de genie

École d'Ingénierie et de
Technologie de l'Information

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

CSI2372A
Advanced Programming Concepts with C++
Exercises

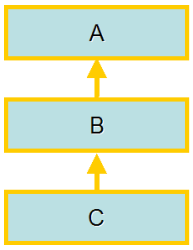
Professor: Jochen Lang

Reminder: The exam will be open book but closed laptop and other electronic devices.

PART 1: MULTIPLE CHOICE QUESTIONS

Select the best answer. There is only one best answer.

1. Destruction of a subclass C. C is a sub-class of B which in turn is a sub-class of A.



   a. The destructor of C destructs C and since C is a sub-class of B (and A), the destructor of B (and A) does need not to execute.
   b. The destructor of C first destructs A then B and then itself (C).
   c. The destructor of C first destructs itself and then the destructor of B is executed, followed by the destructor of A.
   d. The destructor of C uses the same order of destruction than the constructor of C uses for construction.
   e. None of the above

2. An abstract class is a class that

   a. must have no data members.
   b. must have no member functions.
   c. is privately derived from an abstract base class.
   d. has one or more pure virtual functions.
   e. none of the above.

3. Multiple inheritance

   a. Is not possible in C++ and is only available in Java.
   b. Multiple inheritance means that a sub-class can be derived from a class which in turn is derived from another base class.
   c. Multiple inheritance allows a sub-class to have more than one direct non-abstract base class.
   d. Multiple inheritance requires that all but one base class are abstract.
   e. None of the above

4. The best way to create functions that perform identically the same task on different data types is to use:

   a. function overloading.
   b. function subclassing.
   c. function templates.
   d. virtual functions.
   e. none of the above.

5. STL Containers and algorithms

   α. Iterators can only be used with sequential containers (vector, deque and list).
   β. stl::list implements a singly-linked list
   χ. The decrement operator (--) should always be used with a reverse_iterator.
   δ. The generic algorithms in the STL can also be used with pointers instead of iterators.
   ε. None of the above

PART 2: BEHAVIOUR OF C++ PROGRAMS

1. Correct the following program [2]:

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

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

int main() {
  std::unique_ptr<string> aPtr(new string("Hello"));

  std::unique_ptr<string> aPtr2;
  // Transfer ownership from aPtr to aPtr2
  aPtr2 = aPtr;
  aPtr2.reset(aPtr.release());
  cout << *aPtr2 << endl;

  // Transfer ownership from aPtr2 to aPtr3
  std::unique_ptr<string> aPtr3(aPtr2);
  std::unique_ptr<string> aPtr3( std::move(aPtr2));
  cout << *aPtr3 << endl;

  return 0;
}
```

Console Output:

```
Hello
Hello
```

2. What does the following program print [3]:

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

using namespace std;

int main(void)
{
  fstream fin, fout;
  int iVal = 40;
  char cVal, a = 'a';
  string str("a \n40");
  fout.open("test.txt",ios::out);
  fout << a << " \n" << iVal;
  fout.close();
  fout.clear();
  fout.open("test.bin",ios::out | ios::binary);
  fout.write(str.c_str(), str.length());
  fout.close();
  fin.open("test.txt",ios::in);
  fin >> cVal >> a;
  cout << cVal << a << endl;
  fin.close();
  fin.clear();
  fin.open("test.bin",ios::in | ios::binary );
  char buf[5]; // should be char buf[6]; buf[5]=0;
  // in order for buf to be a 0-terminated string
  fin.read( buf, 5 );
  cout << buf; // Needed here
  fin.close();
  return 0;
}
```

a4
a
40

3. The following program makes use of a function template and a class template. Complete the main function *and* specify *all* instantiations for this program in a format suitable for a template instantiation file [7]

```
template <class T, const int WIDTH, const int HEIGHT>
class window {
  T d_array[WIDTH][HEIGHT];
public:
  const T* operator[]( int _width ) const {
    return d_array[_width];
  }

  T* operator[]( int _width ) {
    return d_array[_width];
  }

   void setCenter( T _value ) {
     for ( int h=0; h<HEIGHT; h++ ) {
       for ( int w=0; w<WIDTH; w++ )
         d_array[w][h] = static_cast<T>(0);
     }
     d_array[WIDTH/2][HEIGHT/2] = _value;
     return;
   }
};
template <typename T1, typename T2, const int WIDTH, const int HEIGHT>
  void copy( window<T1,WIDTH,HEIGHT>& w1, window<T2,WIDTH,HEIGHT>& w2 )
{
  for ( int h=0; h<HEIGHT; h++ ) {
    for ( int w=0; w<WIDTH; w++ ) {
      w2[w][h] = static_cast<T2>(w1[w][h]);
    }
  }
  return;
}
int main() {
  // Define a character window of size 12x8
  window<char,12,8> charWindow;
  // Define a double window of size 12x8
  window<double,12,8> doubleWindow;
  // use method setCenter on the double window
  doubleWindow.setCenter( 98.1 );
  // use copy from double window to char
  // All template parameters are inferred by the compiler!
  copy( doubleWindow, charWindow );
  return 0;
}
```

4. The following program uses the move constructor and the move assignment. Add the move constructor and move assignment operator in the class MoveIt. [6]

```
template <class T>
class MoveIt {
  unsigned int d_size;
  T* d_array;
public:
  MoveIt();
  MoveIt( T* _in, unsigned int _size);
  MoveIt(const MoveIt<T>& _otn);
  ~MoveIt();
  MoveIt& operator=(const MoveIt<T>& _otn);
  // Move ctor
   MoveIt( MoveIt<T>&& _otn ) noexcept;
   // Move assignment
   MoveIt<T>& operator=(MoveIt<T>&& _otn) noexcept;

  // other definitions
};

template <class T>
MoveIt<T>::MoveIt( MoveIt<T>&& _otn ) noexcept {
  cerr << "MoveIt(MoveIt<T>&& _otn)" << endl;
  // Stealing all resources from _otn
  d_size = _otn.d_size;
  d_array = _otn.d_array;
  // Leaving _otn in a defined state
   _otn.d_size = 0;
   _otn.d_array = nullptr;
}

template <class T>
MoveIt<T>& MoveIt<T>::operator=(MoveIt<T>&& _otn) noexcept
{
  cerr << "MoveIt& operator=(MoveIt<T>&& _otn)" << endl;
  if (this != &_otn ) {
    // save to delete
    if (d_array) delete[] d_array;
    d_array = _otn.d_array;
    d_size = _otn.d_size;
    _otn.d_size = 0;
    _otn.d_array = nullptr;
  }
  return *this;
}
```

5. The following routine prints the elements of type `int` stored in `std::vector` in-order.

```
void printInOrder( vector<int>&  container ) {
  // loop over the elements
  for ( vector<int>::iterator iter = container.begin();
        iter != container.end();
        ++iter ) {
    // save to access *iter
    cout << *iter << ' ';
  }
  cout << endl;
  return;
}
```

a. Change the above routine to print the elements stored in the container in reverse order [2]

```
void printInReverseOrder( vector<int>&  container ) {
  // loop over the elements
  for ( vector<int>::reverse_iterator iter = container.rbegin();
        iter != container.rend();
        ++iter ) {
    // save to access *iter
    cout << *iter << ' ';
  }
  cout << endl;
  return;
}
```

b. Use the `std::for_each` algorithm to achieve the same result as the given routine [6].

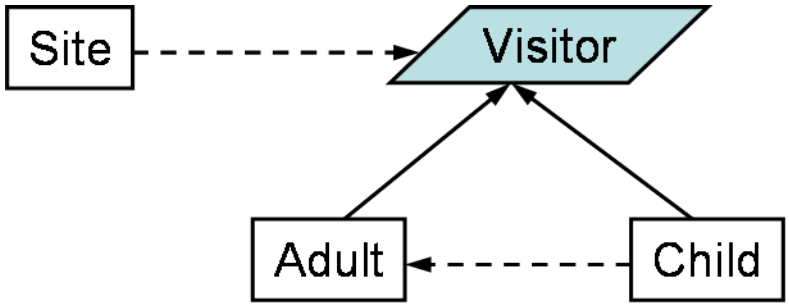
Hint: `for_each` *expects 3 arguments: the first two specifying a range with iterators and the third one a function or functor which takes a single argument which is of the type stored in the container, here:* `int`.

```
template<class T>
void Print( const T& x ) {
  cout << x << ' ';
}

void printInOrder( vector<int>&  container ) {
  for_each( container.begin(), container.end(), Print<int> );
  cout << endl;
  return;
}
```

PART 3: PROGRAMMING QUESTIONS

1. The following program is a rudimentary visitor program at a historic site. The site is visited by adults and children. The site has a membership program, i.e., adult and child visitors may be members. The site administration also likes to keep track of the mode of transportation of the visitors because of parking needs.



- Definition of Site class

```
class Site {
  Visitor** d_allVisitors;
  int d_nextVisitor;
  int d_size;
public:
  Site( int _size ) :
   d_size( _size ), d_nextVisitor( 0 ), d_allVisitors( 0 )
  {
    if ( _size <= 0 )
      throw std::out_of_range( "Site must accompany at least 1 visitor!" );
    d_allVisitors = new Visitor*[d_size];
  }

  Site& operator<<( const Visitor& _visitor ) {
    if ( d_nextVisitor == d_size )
      throw std::out_of_range( "Site is full!" );
    d_allVisitors[d_nextVisitor] = _visitor.clone();
    d_nextVisitor++;
    return *this;
  }

  void print() {
    cout << "Visitors at Site: " << endl << endl;
    for ( int i=0; i<d_nextVisitor; i++ ) {
      d_allVisitors[i]->print();
    }
  }
};
```

- Definition of Visitor, Adult and Child classes

```
enum Transport { PUBLIC, PRIVATE };

class Visitor {
  static string lastMemberId;
  string d_memberId;
  bool d_isMember;
public:
  Visitor( bool _isMember=false, string _memberId="" );
  virtual ~Visitor(){};
  virtual void print() const;
  bool isMember();
  virtual Visitor* clone() const = 0;
};

class Adult : public Visitor {
  Transport d_transport;
public:
  Adult( Transport _transport,  bool _isMember=false, string
_memberId="");
  virtual void print() const;
  virtual Adult* clone() const;
};

class Child : public Visitor {
  int d_age;
  Adult d_responsible;
public:
  Child( int _age, Adult& _responsible );
  virtual void print() const;
  virtual Child* clone() const;
};
```

a. Give the code to initialize `Visitor::lastMemberId` to the string `"A0000"` and explain briefly (1 sentence) where this initialization should take place. [2]

```
std::string Visitor::lastMemberId("A0000");
```

It should be in visitor.cpp

b. Overload the extraction operator (>>) for the removal of the most recently added visitor from a Site. The routine should throw std::underflow_error if the `Site` is already empty [4]

```
Site& operator>>( Visitor& _visitor ) {
  if ( d_nextVisitor == 0 )
    throw std::underflow_error( "Site is empty!" );
  _visitor = d_allVisitors[--d_nextVisitor];
  return *this;
}
```

c. Define (implement) the copy constructor with a deep copy strategy for the class `Site` [4]

```
Site( const Site& _oSite )
  : d_nextVisitor( _oSite.d_nextVisitor ), d_size( _oSite.d_size ),
    d_allVisitors( 0 )
{
  d_allVisitors = new Visitor*[d_size];
  for ( int i=0; i<d_nextVisitor; i++ ) {
    d_allVisitors[i] = _oSite.d_allVisitors[i]->clone();
  }
}
```

d. Define (implement) the assignment operator with a deep copy strategy for the class `Site` [5]

```
Site& operator=( const Site& _oSite ) {
  if ( &_oSite == this ) return *this;
  for ( int i=0; i<d_nextVisitor; i++ ) {
    delete d_allVisitors[i];
  }
  delete d_allVisitors;
  d_nextVisitor = _oSite.d_nextVisitor ;
  d_size = _oSite.d_size;
  d_allVisitors = new Visitor*[d_size];
  for ( int i=0; i<d_nextVisitor; i++ ) {
    d_allVisitors[i] = _oSite.d_allVisitors[i]->clone();
  }
  return *this;
}
```

e. Define (implement) the destructor of the class `Site` [3]

```
~Site() {
  for ( int i=0; i<d_nextVisitor; i++ ) {
    delete d_allVisitors[i];
  }
  delete d_allVisitors;
}
```