University of Toronto
Faculty of Applied Science and Engineering

ECE 244F

PROGRAMMING FUNDAMENTALS

Fall 2014

Final Examination

Examiners: T.S. Abdelrahman and M. Stumm

Duration: Two and a Half Hours

This exam is OPEN Textbook and CLOSED notes. The use of computing and/or communicating devices is NOT permitted.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted.

Work independently. The value of each part of each question is indicated. The total value of all questions is 100.

Write your name and student number in the space below. Do the same on the top of each sheet of this exam book.

Name:
(Underline last name)

Student Number:

Q1:		Q8:	
Q2:		Q9:	
Q3:		Q10:	
Q4:		Q11:	
Q5:		Q12:	
Q6:		Q13:	
Q7:		Q14:	

Total:

Question 1. (12 marks). *General.*

Answer the following questions as indicated. No justification of your answer is required.

(a) (1 mark). Circle the best answer. The main purpose of inheritance in C++ is to:

1. enable modular programming.
2. facilitate code reuse.
3. facilitate conversion of data types.
4. modify the capabilities of a class

(b) (1 mark). Circle one or more answers. An object of a _______ class can be treated as an object of its corresponding ________ class.

1. base, derived
2. derived, base
3. Both 1 and 2
4. None of these

(c) (1 mark). True or **False**? A programmer, who wishes to use inheritance to make a derived class from the class called `Base`, does not need access to the source code for the implementation of `Base` (i.e., `Base.cpp`).

(d) (1 mark). True or **False**? An abstract class is one that has no data member, only function members.

(e) (2 marks). Circle all correct answers. The purpose of using a "visited" flag in the code for traversing a graph is to:

1. Ensure that every node in the graph is visited.
2. To prevent a node from being visited more than once.
3. To ensure that the nodes are visited in the correct order of the traversal.
4. To facilitate the use of recursion to traverse the graph.
5. All of the above.
6. None of the above.

(f) **(1 mark). Circle one answer**. Algorithm A has a time complexity of O(n). Algorithm B has a time complexity of O(log n).

1. Algorithm B is always more efficient then algorithm A.
2. Algorithm B is never more efficient than algorithm A.
3. Algorithm A can sometimes be faster than algorithm B.
4. None of the above.

(g) **(1 mark). Circle all correct answers**. A *collision* in a hash table can occur when?

1. two entries are identical except for their keys.
2. two entries with different data have the exact same key.
3. two entries with different keys have the same exact hash value.
4. two entries with the exact same key have different hash values.

(h) **(2 marks). Write your answer next to each part**. What is the complexity using big-Oh notation of algorithms that have the following run times, where n is the size of the input:

1. $T(n) = \log n + 1000$
2. $T(n) = n \log n + 35 n + 4 n^2$
3. $T(n) = 2n + n \log(n^2) + 5000 \log n$
4. $T(n) = n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$

(i) **(2 marks). Mark each of the following statements as True or False (circle one)**. Consider the order in which the leaves of a binary tree are visited by the three traversals: pre-order, in-order, and post-order.

1. The three traversals visit the leaves in different orders.	**True**	**False**
2. The three traversals visit the leaves in the same order.	**True**	**False**
3. Only pre-order and post-order visit the leaves in the same order.	**True**	**False**
4. Only in-order and post-order visit the leaves in the same order.	**True**	**False**

Question 2. (6 marks). *Objects with Pointers.*

Consider the following declaration and implementation of the class `Name` contained in the file `Name.h`.

```
class Name {
   private:
      string* firstName;
      string* lastName;

   public:
      Name(string fname, string lname);
      Name(const Name & src);
     ~Name();
      void print() const;
};
```

Now consider the implementation of the class in `Name.cpp`.

```
#include "Name.h"
#include <iostream>
using namespace std;

Name::Name(string fname, string lname) {
    firstName = new string(fname);
    lastName = new string(lname);
}

Name::Name(const Name & src) {
    firstName = new string(*(src.firstName));
    lastName  = new string(*(src.lastName));
}

Name::~Name() {
    delete firstName;
    delete lastName;
}


void Name::print() const {
      // This prints the first name, followed by a space, followed
      // by the last name
      cout << *firstName << " " << *lastName << endl;
}
```

Now consider the program below that uses the above Name class.

```
#include <iostream>
#include <string>
#include “Name.h”
using namespace std;


int main() {
    string Tom = “Tom”;
    string Johns = “Johns”;
    string Michael = “Michael”;
    string Jackson = “Jackson”;
    string Sarah = “Sarah”;
    string Johnson = “Johnson”;

    Name first(Tom, Johns);
    Name second(Michael, Jackson);
    Name third(Sarah, Johnson);

    first.print();
    second.print();
    third.print();

    first = second;
    second = third;
    third = first;

    first.print();
    second.print();
    third.print();

    // Point A

    return (0);
}
```

(a) **(1 mark)**. What is the output produced by the program above when it reaches `Point A`?

(b) **(1 mark)**. Does the program above suffer from any memory leaks by the time it reaches point A? If so, then how many variables (`ints`, `strings`, or objects of type `Name`) end up being memory leaks?

(c) **(1 mark)**. The program above also suffers another problem after continues past `Point A`. Describe this problem in one sentence.

(d) **(3 marks)**. The problems with the above implementation of `Name` that can be fixed by the addition of one member function. In the space below, write the implementation of this function. Make sure to include the function header and the function body.

Question 3. (6 marks). *Memory Leaks.*

Assume the size of an integer is 4 bytes, the size of an integer pointer is 8 bytes and the size of a `Node` pointer is also 8 bytes. Now consider the following code:

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

class Node {
  private:
    int* value;
    Node* next;
  public:
    Node(int a) { value = new int; *value = a; next = NULL;}
    void setValue(int a){ *value = a; }
    int getValue(){ return *value; }
    void setNext(Node* n){ next = n; }
    Node* getNext(){ return next; }
    void print(){
        cout << *value << endl;
        if(next != NULL) next->print();
    }
};

int main(){
    int i = 0;
    Node* p;
    Node t1(i);
    p = &t1;
    i++;

    for(; i < 100; i++) {
        Node t2(i);
        p->setNext(&t2);
        p = &t2;
    }
    // Point A
    t1.print();
    return 0;
}
```

(a) (2 marks). How much memory (in bytes) has been leaked at `Point A` in the program?

(b) (2 marks). How much memory (in bytes) has been leaked upon returning from `main()`?

(c) (2 marks). What happens when `print()` in the above program is executed?

Question 4. (5 marks). *Functions.*

Consider the following code of a class `Foo`. You may assume that the code is error free.

```
class Foo {
  private:
     int x;

  public:
     Foo(int i);
     int getX() const;
     void setX(int i);

};


Foo::Foo(int i) {
    x = i;
}

int Foo::getX() const {
    return x;
}

void Foo::setX(int i) {
    x = i;
}
```

Consider the following non-member function `printNegative`:

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

void printNegative(const Foo & source) {
   cout << -1*source.x << endl;
}
```

(a) **(1 mark).** This function, as written, has a problem with it. Describe what the problem is (one short sentence).

(b) **(2 marks).** If we do not wish to change the function nor make it a member function of `Foo`, what change must be made to the class `Foo` to make `printNegative` correct? You may write your answer in the code of `Foo` above.

(c) **(2 marks).** If we can change the function but still not make it a member function of `Foo`, re-write the body of the function to make it correct? Write your answer below.

Question 5. (4 marks). *Operator Overloading.*

One of the disadvantages of C and C++ arrays is that they have no bounds checking. This results in erroneous program behavior and crashes when the program contains off-by-one errors when indexing into the array. For this reason, you intend to implement a "smart array" of pointers to objects of type `T`. The intent is to automatically check the bounds of the array when indexing into the array. The declaration of your smart array starts as follows:

```
class SmartTarray {
  private:
    T **array;
    int size;
  public:
    SmartTarray( int arraySize ) ;
    ...
};
```

You may assume that the class `T` and the constructors/destructor of the class `SmartTarray` are properly implemented. Write the `operator[]` function that is used to index into the array and returns the indexed pointer to the `T` object if successful. If the array is accessed with an index out of bounds, it should output the error message "`out of bound access`" and return a `NULL` pointer.

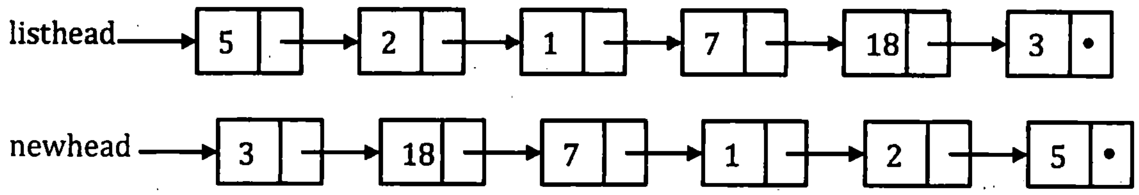
Write your answer in the box below.

Question 6. (7 marks). *Recursion and Linked Lists.*

Consider the following class definition of nodes of a linked list.

```
class listNode {
    private:
        int key;
        listNode* next;
    public:
        // Constructors & Destructor
        :
        // Accessors
        int getKey();               // Returns key of node
        listNode* getNext();        // Returns pointer to next node
        // Mutators
        void setKey(int k);
        void setNext(listNode* n);
};
```

Write a **recursive** non-member function `listNode* ReverseList(listNode* head)` that reverses the linked list in place, i.e., without using an array or creating another linked list, and returns a pointer that is the new head of the list. Thus, in the example below, the invocation of the function as `newhead = ReverseList(listhead)` reverses the list as shown.



Write your answer in the box below.

```
listNode* ReverseList(listNode* head) {










}
```

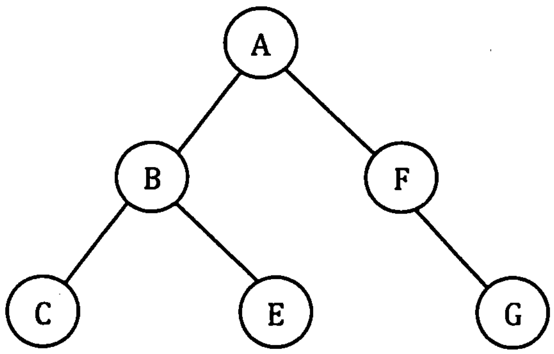
Question 7. (6 marks). *Tree Traversals*

Consider the following class definition of a tree node.

```
class treenode {
   public:
     int key;
     treenode* left;
     treenode* right;
};
```

Write a non-member function to perform the *2-left pre-order* traversal of a binary tree, meaning that for each node of the tree, the function always first visits the node, traverses the left subtree (in 2-left pre-order), traverses the right subtree (in 2-left pre-order), then traverses the left subtree again (also in 2-left pre-order). Visiting a node simply prints its key to `cout`.

(a) (2 marks). Give the 2-left pre-order traversal of the following tree.



Give the traversal here:

(b) (4 marks). Write the function here. Your code should be very short.

```
void tl_preorder(treenode* root) {









}
```

Question 8. (5 marks). *Binary Trees.*

Assume that binary trees are implemented using the following declarations:

```
class TreeNode{
  private:
    int value;
    TreeNode * left;
    TreeNode * right;

  public:
    TreeNode() ;
    TreeNode( int v ) ;
    ~TreeNode() ;
    void insert( int v ) ;
    bool search( int v ) ;
    int count( int v ) ;
    ...
};

class Tree{
  private:
    TreeNode * root ;
  public:
    Tree() ;
    ~Tree() ;
    void insert( int v ) {
        if( root==NULL ) root = new TreeNode( v ) ;
        else root->insert( v ) ;
    }
    bool search( int v ) {
        if( root == NULL ) return false ;
        return root->search( v ) ;
    }
    int count( int v ) {
        if( root == NULL ) return 0 ;
        return root->count(v) ;
    }
    ...
};
```

Implement the function `TreeNode::count(int v)` that returns a count of the number of nodes with value equal to `v`:

```
int TreeNode::count(int v) {




}
```

Question 9. (7 marks). *Binary Search Trees.*

Consider the following definition of a node in a binary tree.

```
class treeNode {
    private:
        int key;
        treeNode* leftChild;
        treeNode* rightChild;
    public:
        // Constructors & Destructor
        :
        // Accessors
        int getKey();           // Returns key of node
        treeNode* getLeft();    // Returns pointer to left subtree
        treeNode* getRight();   // Returns pointer to right subtree
        treeNode* getMin();     // Returns pointer to node with min key
                                //     in the tree rooted at this node
        treeNode* getMax();     // Returns pointer to node with max key
                                //     in the tree rooted at this node
        // Mutators
        :
};
```

Implement a <u>non-member</u> function `bool isBST(treeNode* root)` that takes a pointer to a binary tree rooted at `root` and returns true if this binary tree is also a binary search tree or false if otherwise. Write your answer below.

```
bool isBST(treeNode* root) {

}
```

Question 10. (5 marks). *Graphs.*

Given the following group of five people: James, Scott, Steve, Linda and Sara. The following pairs of people are acquainted with each other.

James and Steve
James and Linda
Scott and Steve
Steve and Linda
Steve and Sara
Linda and Sara

(a) (2 marks). Draw a graph that represents the five people and their acquaintance.

(d) (3 marks). Write a traversal of the graph in part (a) above that (i) starts at the node "James", (ii) visits children before siblings.

Question 11. (12 marks). *Inheritance.*

Consider the following three classes, `Base`, `Intr` and `Derived`.

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

using namespace std;

class Base {
  private:
    string* b;
  public:
    Base();
    Base(string n);
    Base(const Base & t);
    virtual ~Base();
    void print();
};

Base::Base() {
 b = new string ("B");
 cout << "Base 1: " << *b << endl;
}


Base::Base(string n) {
 b = new string (n);
 cout << "Base 2: " << *b << endl;
}

Base::Base(const Base & t) {
 b = new string (*t.b);
 cout << "Base 3: " << *b << endl;
}

Base::~Base() {
 cout << "Base -1: " << *b << endl;
 delete b;
}

void Base::print() {
 cout << "Base 0:"  << *b <<  endl;
}
```

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

using namespace std;

class Intr : public Base {
   private:
     string* i;
   public:
     Intr();
     Intr(string n, string m);
     Intr(const Intr & t);
     virtual ~Intr();
     virtual void print();
};

Intr::Intr() {
 i = new string ("I");
 cout << "Intr 1: " << *i << endl;
}

Intr::Intr(string n,
           string m):Base(n) {
 i = new string (m);
 cout << "Intr 2: " << *i << endl;
}

Intr::Intr(const Intr & t) {
 i = new string (*t.i);
 cout << "Intr 3: " << *i << endl;
}

Intr::~Intr() {
 cout << "Intr -1: " << *i << endl;
 delete i;
}

void Intr::print() {
 Base::print();
 cout << "Intr 0: " << *i <<  endl;
}
```

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

using namespace std;

class Derived : public Intr {
   private:
     string* q;
   public:
     Derived();
     Derived(string n, string m, string q);
     Derived(const Derived & t);
     virtual ~Derived();
     virtual void print();
};

Derived::Derived() {
  q = new string ("D");
  cout << "Derived 1: " << *q << endl;
}

Derived::Derived(string n, string m, string k):Intr(n,m) {
  q = new string (k);
  cout << "Derived 2: " << *q << endl;
}

Derived::Derived(const Derived & t) {
  q = new string (*t.q);
  cout << "Derived 3: " << *q << endl;
}

Derived::~Derived() {
  cout << "Derived -1: " << *q << endl;
  delete q;
}

void Derived::print() {
  Intr::print();
  cout << "Derived 0: " << *q <<  endl;
}
```

These classes will be used to answer the questions on the following page.

(a) **(3 marks)**. The following is a `main` function that uses the above three classes. The code is shown in a table to facilitate writing your answers. Place an `X` in the second column of the table next to line in this code that generates a compile-time error.

Code	Compile-Time Error?
`using namespace std;`	There are no errors generated by these lines
`#include <iostream>`	
`#include <string>`	
`int main() {`	
`string one = "one";`	
`string two = "two";`	
`string three = "three";`	
`Base a;`	
`Intr b;`	
`Derived c;`	
`Base* pbase = &a;`	
`Intr* pinterm = &b;`	
`Derived* pderived = &c;`	
`b = c;`	
`b = a;`	
`a = c;`	
`pbase = pderived;`	
`pderived = pinterm;`	
`pinterm = pbase;`	
`return (0);`	There are no errors generated by these lines
`}`	

(b) **(5 marks)**. The following is a `main` function that uses the above three classes. The code is shown in a table to facilitate writing your answers. Indicate the output generated when each line of the code is executed. If no output is generated, leave the entry in the table blank. There are no compile-time errors in the code.

Code	Output

`using namespace std;`
`#include <iostream>`
`#include <string>`
`int main() {`
	`Provide no answer for these lines`

`Base a("one");`	
`Intr b("one", "two");`	
`Intr c(b);`	
`Derived d("one","two","three");`	
`return (0);`	
`}`	

(c) **(4 marks)**. The following is a `main` function that uses the above three classes. The code is shown in a table to facilitate writing your answers. Indicate the output generated when each line of the code is executed. If no output is generated, leave the entry in the table blank. There are no compile-time errors in the code.

Code	Output
`using namespace std;`	Provide no answer for these lines
`#include <iostream>`	
`#include <string>`	
`int main() {`	
`Base* pb = new Intr("one", "two");`	
`Intr* pi = new Derived("one","two","three");`	
`pb->print();`	
`pi->print();`	
`delete pb;`	
`delete pi;`	
`return (0);`	Provide no answer for these lines
`}`	

Question 12. (9 marks). *Virtual operators.*

Declaring a function within a class as "virtual" enables polymorphism and late binding, where the particular function that is invoked depends on the underlying object type. Operators (e.g., `operator*`) can also be declared as virtual functions, just like other member functions. However, operators can cause extra complexity because they may operate on two objects, both with unknown types.

Consider, for example, implementing `operator*` in a linear algebra library so that it can be used to multiply any combination of matrix, vector, and/or scalar objects. The code below shows how this can be done, even though it may appear to be somewhat convoluted. In the code below, an abstract class called `LAAC` (for "Linear Algebra Abstract Class") is declared, and the `Matrix`, `Vector` and `Scalar` classes are derived from `LAAC`. You will notice that `operator*` is declared as virtual, and each of the subclasses implements their own `operator*` function. However, the code for each `operator*` implementation is exactly the same. In each case, the implementation calls a second virtual function, `multiply()`, on `operator*`'s argument. This second function is used to late-bind the second argument

```
class Matrix;
class Scalar;
class Vector;

class LAAC {
  public:
    virtual LAAC& operator*(LAAC& rhs) = 0;
    virtual LAAC& multiply(Matrix*) = 0;
    virtual LAAC& multiply(Scalar*) = 0;
    virtual LAAC& multiply(Vector*) = 0;
    virtual ~LAAC() {}
};

class Matrix : public LAAC {
public:
  LAAC& operator*(LAAC& rhs) {  return (&rhs)->multiply(this); }
  LAAC& multiply(Matrix*) ;
  LAAC& multiply(Scalar*) ;
  LAAC& multiply(Vector*) ;
};

class Scalar : public LAAC {
public:
  LAAC& operator*(LAAC& rhs) {  return (&rhs)->multiply(this); }
  LAAC& multiply(Matrix*) ;
  LAAC& multiply(Scalar*) ;
  LAAC& multiply(Vector*) ;
};

class Vector : public LAAC {
public:
  LAAC& operator*(LAAC& rhs) {  return (&rhs)->multiply(this); }
  LAAC& multiply(Matrix*) ;
  LAAC& multiply(Scalar*) ;
  LAAC& multiply(Vector*) ;
};
```

In the table below, indicate which specific operation does each of the following functions have to implement. You can indicate the specific operation by using the following notation: "s*M" for scalar-by-matrix multiplication, "M*v" for matrix-by-vector multiplication, or "v*s" for vector-by-scalar multiplication.

Function	Operation
`LAAC& Matrix::multiply( Matrix *)`	
`LAAC& Matrix::multiply( Vector *)`	
`LAAC& Matrix::multiply( Scalar *)`	
`LAAC& Vector::multiply( Matrix *)`	
`LAAC& Vector::multiply( Vector *)`	
`LAAC& Vector::multiply( Scalar *)`	
`LAAC& Scalar::multiply( Matrix *)`	
`LAAC& Scalar::multiply( Vector *)`	
`LAAC& Scalar::multiply( Scalar *)`	

Question 13. (9 marks). *Complexity Analysis.*

(a) **(2 marks)**. Give the time complexity of the following segment of code, where n is the size of the input. Show your analysis for part marks.

```
for (int i = 0; i < n; ++i)
   for (int j = i; j < n; ++j)
     for (int k = 0; k < n; ++k)
         O(1)
```

```
T(n) = O (        )
```

(b) **(2 marks)**. Give the time complexity of the following segment of code where n is the size of the input. Show your analysis for part marks.

```
for(int i=0; i<n; i++)
   for(int j=n; j>0; --j)
      for(int k=0; k<n*n; k++)
         O(1)
```

```
T(n) = O (        )
```

(c) (2 marks). Give the time complexity of the following segment of code where n is the size of the input. Write down the recurrence equation and show your analysis for part marks.

```
void recursive(int n) {
   if (n == 1) return;
   recursive (n-1);
}
```

Recurrence equation: T(n) =

T(n) = O (   )

(d) (3 marks). Give the time complexity of the following segment of code where n is the size of the input. Write down the recurrence equation and show your analysis for part marks.

```
int recursive(int n) {
   if (n == 1) return (1);
   O(n)
   return (recursive (n/3) + 2*recursive(n/3) + 3*recursive(n/3));
}
```

Recurrence equation: T(n) =

T(n) = O (   )

Question 14. (7 marks). *Analysis of Algorithms.*

A ***heap*** is an array-based data structure not covered in class. A heap has two primary operations: `insert(key)` and `removeMax()`. The `removeMax` function returns the object with the highest valued key. The following code is a possible implementation, where we only store keys and no other data:

```
class Heap {
 private:
  int capacity;
  int *A;
  int storedElements ;

public:
  Heap(int n) {
    capacity = n ;
    A = new int[ capacity ] ;
    storedElements = 0 ;
  }
  ~Heap() ;
  bool insert( int key ) ;
  bool removeMax( int & max ) ;
};

bool Heap::insert( int key ) {
    if( storedElements == capacity ) return false ;
    int i = storedElements++ ;
    A[i] = key ;
    while( (i>0) && (A[(i-1)/2] < A[i]) ) {
        swap( A[(i-1)/2], A[i] ); // Swaps the two array elements in constant
time
        i = (i-1)/2 ;
    }
    return true ;
}

bool Heap::removeMax(int &max) {
    if( storedElements == 0 ) return false ;

    max = A[0] ;
    storedElements-- ;
    A[0] = A[storedElements] ;
    int i = 0 ;
    while( i*2+1 < storedElements ) {
        int c = i*2+1 ;
        if( c < storedElements + 1 )
            if( A[c] < A[c+1] )
                c++ ;
        if( A[i] < A[c] ) swap( A[i], A[c] ) ;
        i = c ;
    }
    return true ;
}
```

Using big-Oh notation, what is worst-case running time of the heap `insert` function?

In two or fewer lines, explain your above answer:

Using big-Oh notation, what is worst-case running time of the heap `removeMax` function?

In two or fewer lines, explain your above answer:

In principle, a binary search tree could have been used instead of this heap. Using big-Oh notation, what would the worst case running time of a `removeMax()` function implemented for a binary search tree be?