C++ Final Exam 2016

09:00 - 10:30

Name: ____________________

Student id: _________________

Important:
This exam is closed book. You are only allowed to use a dictionary (paper or electronic) as additional material.
Any other electronic devices (computer, cell-phones, tablet, …) are forbidden.

Problem 1: Inheritance
Implement an interface “Shape3D” with a single method “double ComputeVolume()” (this method will compute the volume of a given shape). Assume that “double” precision floating point types are used everywhere for representing numbers.
Implement the concrete sub-classes of "Shape3D": “Cube” and "Ball". A cube can be represented by its lowest vertex, and the cube's length. A ball can be represented by its center and radius.

Answer 1:
```
#define _USE_MATH_DEFINES
#include <cmath>

class Shape3D {
public:
 virtual double ComputeVolume() = 0;
};

class Cube: public Shape3D {
public:
 Rectangle (double x, double y, double z, double length) : cx(x), cy(y), cz(z), length(length) {}
 virtual double ComputeVolume() { return length*length*length; }

private:
 double cx,cy,cz,length;
};

class Sphere: public Shape3D {
public:
 Sphere(double x, double y, double z, double radius) : cx(x), cy(y), cz(z), radius(radius) {}
 virtual double ComputeVolume() { return 4.0/3.0*M_PI*radius*radius*radius; }
private:
 double cx,cy,cz,radius;
};
```

Problem 2:
Question 2.1: What is the keyword “friend” used for in C++?

Answer 2.1:
To grant permission to functions or classes to access private fields of a given class.

Question 2.2: Is it possible to overload the operator: "operator[] (int i, int j)" to access elements of a two-dimensional array? Why?

Answer 2.2: No. Because [] is unary.

Question 2.3: Is it possible to define “operator**” corresponding to the “to-the-power-of” operation? Why?

Answer 2.3: No. Because there is no built-in ** operator (only existing operators can be overloaded).

Problem 3: Generic programming
Question 3.1: Write a template function "adjacent_find" that given a pair of iterators "first" and "last", searches the range ["first"; "last") for two consecutive identical elements.
This function returns the first element in the first pair of identical elements. If no such pair is found, then it returns "last".
Assume that comparisons are done with "==".

Answer 3.1:
```
template<class T>
T adjacent_find(T first, T last) {
  if (first == last) return last;
  T next = first; next++;
  for (; first != last; ++first, ++next) {
    if (*first == *next) return first;
 }
 return last;
}
```

Problem 4: Standard library.
The goal of this question is to write a program to remove duplicates from a long list of persons.
Persons are represented by the following class:

```
struct Person {
 std::string first_name;
 std::string last_name;
};
```

To remove the duplicates you will write a function that relies on a hash table ("unordered_set").

Question 4.1: Hash function and equality.
In order to use instances of "Person" with an "unordered_set", define an hash function and equality function for the struct "Person".
Two "Person"s are equal if they have the same first name and last name.
For the hash function use the following expression: 11*hash(first name) + 7*hash(last name), where hash() is a hash function for string (you can use the one provided by the standard library).

Answer 4.1:

```
struct HashPerson {
 std::size_t operator() (const Person& p) const {
  return 11*std::hash<std::string>()(p.first_name) + 7*std::hash<std::string>()(p.last_name);
}
};

struct HashEqual {
 bool operator() (const Person& p1, const Person& p2) const {
   return p1.first_name == p2.first_name && p1.last_name == p2.last_name;
}
};
```

Question 4.2: Remove duplicate.
Write a function "std::vector<Person> RemoveDuplicates(std::vector<Person>& persons)" that takes as input a vector of persons and return as output a vector of persons with the duplicate entries removed.
To write this function use an "unordered_set" to remove duplicates.

Answer 4.2:

```
std::vector<Person> RemoveDuplicates(std::vector<Person>& persons) {
  std::unordered_set<Person, HashPerson, EqualPerson> s(persons.begin(), persons.end());
  return std::vector<Person>(s.begin(), s.end());
}
```

Problem 5: Manipulation of arithmetic expression.
In this problem we consider arithmetic expression constructed from constant, addition and multiplication. For example, the expression: 2 * (3 + 4).
We will assume that numbers are represented by integers everywhere.
We start from the class "Expr" (currently empty) to represent such expressions:

```
class Expr {
public:
 virtual ~Expr() {}
};
```

Question 5.1: Define three subclasses of "Expr": "Const", "Add" and "Mul" and their constructor.
They should allow us to define the example above as:
Expr* e = new Mul(new Const(2), new Add(new Const(3), new Const(4)))

Answer 5.1:

```
class Const: public Expr {
 public:
  Expr(int n): n(n) {}
 private:
  int n;
};
class Add: public Expr {
 public:
  Add(Expr* e1, Expr* e2) : e1(e1), e2(e2) {}
  ~Add() { delete e1; delete e2; }
Private:
 Expr* e1;
 Expr* e2;
};
```

Mul is defined similarly.

Question 5.2: Define the (pure virtual) method "int Eval()" in "Expr" and the corresponding overridden methods in the three sub-classes "Const", "Add" and "Mul".
Remark: You do not need to rewrite all the code for the classes "Expr", "Const", "Add" and "Mul", but you only need to indicate the changes.

Answer 5.2:

```
class Expr {
…
virtual int Eval() = 0;
};
class Const : public Expr {
…
virtual int Eval() { return n; }
};
class Add : public Expr {
…
virtual int Eval() {return e1->Eval() + e2->Eval();
}
class Mul : public Expr {
…
virtual int Eval() { return e1->Eval() * e2->Eval();
}
```

Question 5.3: Postfix notation.
The postfix notation for an expression, also sometimes called Reverse Polish Notation and abbreviated by RPN, is a way of writing an arithmetic expression without parentheses by giving the operators after the operands.
For example, 2 * (3 + 4) becomes in RPN: 2 3 4 + *
In this question, we will represent an RPN expression by a "std::vector<std::string>", where each string is either an operator "+" or "*" or a number (for example "7" or "-3").
Define the method "std::vector<std::string> ToRPN()" in the class "Expr".
This method "ToRPN()" returns a postfix version of the given expression.
For example, if "e" represents the expression 2*(3+4), then "e->ToRPN()" will return the following list: ("2", "3", "4", "+", "*").

Remark:
Assume that you are given a function: "std::to_string(int e)" that converts a given integer into a string [Such a function is actually provided by C++-11].

Answer 5.3:

```
class Expr {
…
virtual void ToRPN(std::vector<std::string>& rpn) = 0;
std::vector<std::string> ToRPN() {
  std::vector<std::string> l;
  ToRPN(l);
  return l;
}
};

class Const: public Expr {
…
virtual void ToRPN(std::vector<std::string>& rpn) {
  rpn.push_back(std::to_string(n));
}
};

class Add: public Expr {
…
virtual void ToRPN(std::vector<std::string>& rpn) {
   e1->ToRPN(rpn);
   e2->ToRPN(rpn);
   rpn.push_back("+");
 }
};
```

Same for Mul.

==
Extra question. Removed.

Problem: Standard library
With the help of the standard library of C++, write a program that takes a "std::string" as input and tests if the letters of the word can be permuted to form a palindrome.
For example, the letters of "deified" can be permuted to form "edified", which is a palindrome, so the function would returns "true".
The idea is that the letters can be permuted to form a palindrome if the count of each letter is even, except for, at most, one letter (for example: “f” in “deified”).
There are different solutions to this problem trading time for space. You need to implement only one solution, but you should discuss the alternatives.
You can assume that the input string is already in lower-case.

Remark: Assume that you have access to a compiler implementing the C++-11 standard.

Solution:
Two possible approaches:
* sort in place, then check that there is no more than one character with an odd count.

* use a hash map, mapping char to their count, and check that there is no more than one character with an odd count.

The first solution has O(n log n) time complexity and O(1) space complexity.
The second solution has O(n) time complexity (on average) and O(c) space complexity.
n is the length of the string. c is the number of distinct characters.
Note that if we did not have a C++-11 compiler, the map based approach would have complexity O(n log n) and space O(c), because map are implemented with balanced binary tree.

Below I give an implementation for both approaches:

```
bool can_form_palindrome(std::string& s) {
  std::sort(s.begin(), s.end());
  int odd_count = 0;
  unsigned int j = 0;

  for (unsigned int i = 1; i < s.size(); i=i+1+j) {
    j = 0;

    while (s[i+j] == s[i-1]) {
      j++;
    }
    if ((j+1)&1) {
      odd_count++;
    }
  }

  return odd_count <= 1;
}


bool can_form_palindrome(const std::string& s) {
  std::unordered_map<char, int> m;
  for (unsigned int i = 0; i < s.size(); ++i) {
    m[s[i]]++;
  }
  int odd_count = 0;
  for (auto it = m.begin(); it != m.end(); ++it) {
    if (it->second & 1) odd_count++;
  }
 return odd_count <= 1;
}
```

Remark:
An alternative is to consider an array of size 26 to represent the map. Char ‘a’ is at position 0, ‘b’ at the position 1, etc