LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming Examination

2016–03–19

1. a) Compilation error. In `print`: the output operator isn't defined for objects of class `A`. It must be defined outside class A as a global function. In order to access the size of the array within A the output operator must be declared as a friend of A. Alternatively, you could define a public member function `size()` in A that returns the size.

```
/* In definition of class A: */
friend std::ostream& operator<<(std::ostream& os, const A& a);

/* As a global function (outside class A). */
std::ostream& operator<<(std::ostream& os, const A& a) {
    for(std::size_t i = 0 ; i != a.size ; ++i) {
        if (i > 0) {
            os << ", ";
        }
        os << a[i];
    }
}
```

   b) Memory leak. The class allocates dynamic memory that never is released. Implement a destructor which deletes the memory:

```
A::~A() {
    delete [] storage;
}
```

   In `print`, the parameter `a` is called by value, and the default copy constructor transmits the value. It only copies the pointer. You must implement a copy constructor which performs a deep copy:

```
A::A(const A& a) {
    size = a.size;
    storage = new int[size];
    for(std::size_t i = 0 ; i != size ; ++i) {
        storage[i] = a.storage[i];
    }
}
```

   In `a2 = a1` the default assignment operator is used. It only copies the pointer and doesn't delete the storage in `a2`. You must implement an assignment operator which deletes the storage and performs a deep copy. Write the code in such a way that it handles self-assignment correctly:

```
A& A::operator=(const A& a) {
    int *new_storage = new int[size];
    size = a.size;
    for(std::size_t i = 0 ; i != size ; ++i) {
        new_storage[i] = a.storage[i];
    }
    delete [] storage;
    storage = new_storage;
    return *this;
}
```

c) The ”rule of five” says that if a class needs any of the five copy control members destructor, copy constructor, assignment operator, move constructor, or move assignment operator, it should probably implement all of them.

d) If the destructor is not virtual, the destructor in the class the pointer variable in the call to `delete` is an instance of execute. We will thus not necessarily execute the destructor in the class the object actually is an instance of. Example:

```
class A {
public:
    ...
    ~A() { }
    ...
};

class B : public A {
public:
    B() : b(new int[10]) {}
    ~B() { delete [] b; }
    ...
private:
    int *b;
};

void f() {
    A* a = new B();
    ...
    delete a;
}
```

When `delete a;` is run, the destructor in `A` will execute and the array `b` will never be deallocated. If the destructor is virtual, the destructor in B will execute.

2. The variable `a` is a pointer and `b` is a reference, so the assignments work as expected, but `c` is a `Bar` object.
   The two first functions are called by reference, so we get the polymorfic behaviour.
   In `print3` a copy of the parameter is made and the copy becomes a `Foo` object (slicing).

```
Bar
Qux
Bar

Bar
Qux
Bar

Foo
Foo
Foo
```

3. Virtual inheritance can occur in connection with multiple inheritance. Suppose we have a class D that is a subclass to both class B and class C. Class B and C are in turn both subclasses to class A. This would cause members from class A to be inherited twice into class D (via B and C respectively). I order to avoid this we can specify that class B and C inherits virtually from class A, stating that they are willing to share members inherited from A in case they would otherwise be inherited twice by subclasses to B and C. In C++ this can be written as:

```
class A {
    ...
};

class B : public virtual A {
    ...
};

class C : public virtual A {
    ...
};

class D : public B, public C {
    ...
};
```

4.
```
class Sensors {
public:
    Sensors() {}
    void update(std::string url);
    double getTemp(const std::string& id) const;
    void print() const;
private:
    std::map<std::string,double> values;
};

void Sensors::update(std::string url) {
    try {
        URLstream is(url);
        std::string xml((std::istream_iterator<char>(is)),std::istream_iterator<char>());
        std::size_t pos = xml.find("<ROMId>");
        while (pos!=std::string::npos) {
            pos += 7;
            std::size_t pos2 = xml.find("</ROMId>",pos);
            std::string romid(xml,pos,pos2-pos);
            pos = xml.find("<Temperature Units=\"Centigrade\">",pos2)+32;
            pos2 = xml.find("</Temperature>",pos);
            std::string temp(xml,pos,pos2-pos);
            double t = stod(temp);
            values[romid] = t;
            pos = xml.find("<ROMId>",pos2);
        }
    } catch (...) { }
}

double Sensors::getTemp(const std::string& id) const {
    if (values.count(id)==0) {
        throw std::runtime_error("Unknown sensor.");
    }
    return values.at(id);
}

void Sensors::print() const {
    std::for_each(values.begin(),values.end(),
        [](std::pair<std::string,double> v) {
            std::cout << v.first << " " <<
            v.second << std::endl; });
}
```

5. a)
```
remove_copy_if(a, a + SIZE, b, [&c](int x) {return x<c;});
```

b)
```
remove_copy_if(a, a + SIZE, ostream_iterator<int>(cout, "\n"),
          [&c](int x) {return x<c;});
```

c)
```
template<typename InIt, typename OutIt, typename Predicate>
OutIt remove_copy_if(InIt first, InIt last, OutIt dest, Predicate pr) {
    for (; first != last; ++first) {
        if (! pr(*first)) {
            *dest++ = *first;
        }
    }
    return dest;
}
```