CS246 Winter 2008— Final Exam

Thursday, 4 April 2008
Instructor: M. W. Godfrey

Time allowed: 150 minutes
No aids allowed (*i.e.,* closed book).

There are 6 questions for a total of K marks.
Plan your time wisely.
In the immortal words of the late Douglas Adams, don't panic!

Answer all of the questions on this exam sheet.
Use the blank exam booklets ONLY as scratch paper.

Name: SOLUTION
(Underline your last name)

Student Number:

(For marking purposes only)

1. **[20 total marks]** *True / false*

For each statement, circle **T** (true) or **F** (false), or do nothing. You get 2 points for a correct answer, 0 points for no answer, and -1 for a wrong answer. However, you cannot get less than 0 on this page.

You may provide a short explanatory statement to support your answer if you desire, but it is not required.

(a) **T** [✓ **F**] If a method is not declared as `virtual` in a parent class, then it's illegal to override it in a child class.

(b) **T** [✓ **F**] If a class embodies a aggregation relationship, then its destructor should almost certainly call the destructor of all of its heap-based aggregate parts.

(c) **T** [✓ **F**] If your code might encounter an exception being thrown, then you should use a "smart" pointer class such as `auto_pointer` to refer to stack-based objects to ensure that they don't become memory leaks.

(d) **T** [✓ **F**] The time complexity of performing an append onto the end of an instance of `std::list` is best described as "amortized constant time".

(e) [✓ **T**] **F** An implementation-only class is one in which the constructors are public, but all other methods (esp. inherited ones) are made private.

(f) [✓ **T**] **F** Copying a composite involves making a "deep copy" of the object's structure and parts.

(g) **T** [✓ **F**] Suppose class `C` inherits from class `P`, and that both provide a fully defined method named `m()`. According to what we learned in class, the precondition of `C::m()` should be stronger than (or the same as) that of `P::m()`.

(h) [✓ **T**] **F** If the destructor of a class is declared as private, it's likely because the designer intended for no client to create objects of that class.

(i) **T** [✓ **F**] In C++, `std::string::operator==` uses object-identity semantics as its model.

(j) [✓ **T**] **F** If a method is declared as virtual, then static dispatch will be used to resolve the call for an object on the stack.

2. **[22 total marks]** *Short answer*

(a) **[2 marks]** What is a memory leak?

ANS: *A memory leak is a ”dead” object that was allocated on the heap but whose storage was never returned to the heap when the object’s use ended. There are no longer any pointers or references to the object; it is completely inaccessible to the running program.*

(b) **[2 marks]** Consider the code below. What is the general term for this? (I’m not referring to exception handling.)

```
void f (Parent* p) {
    Child* c = dynamic_cast<Child*>(p);
    if (c==NULL) {
        // do something drastic, like throw an exception
    } else {
        // happily treat the object as a Child
        // ...
    }
}
```

ANS: *It’s called downcasting.*

(c) **[2 marks]** This phenomenon (*i.e.,* mentioned in part 2b) played a particularly prominent role in the discussion of one design pattern. Name the design pattern in question.

ANS: *It’s a big deal for the Composite pattern.*

(d) **[2 marks]** If a class has a non-virtual destructor, what does this suggest about how you might use the class in your code?

ANS: *It strongly suggests that you should not create a class that inherits from it.*

(e) **[2 marks]** Fill in the blank: A software unit exhibits ________________ to other units when changes to those units are likely to affect it.

ANS: *Tight coupling*

(f) **[4 marks]** Name two kinds of inter-class dependencies that are *not* also navigabilities.

ANS: *Inherits, calls, uses (as a type), instantiates, throws (which is usually also an instantiation), catches.*

(g) **[2 marks]** Briefly state the difference between *data coupling* and *message coupling*.

ANS: *Data coupling is invoking methods that have parameters; message coupling is invocation of messages that do not have parameters.*

(h) **[2 marks]** What is the inheritance relationship between the STL's `vector`, `list`, and `deque`? (Circle the correct answer below.)

i. `vector` and `deque` inherit from `list`.
ii. `deque` and `list` inherit from `vector`.
iii. `vector`, `deque`, and `list` are inheritance cousins, as they all inherit from a common abstract base class.
iv. While the APIs for `vector`, `deque`, and `list` look very similar, there is no actual inheritance relationship bewteen them.

ANS: *2(h)iv*

(i) **[4 marks]** Consider the following class definition:

```
class Balloon {
public :
    Balloon (const std::string& colour);
    ~Balloon();
private :
    std::string colour;
};
Balloon::Balloon(const std::string& colour) : colour(colour) {}
Balloon::~Balloon() {}
```

Extend this class in the space below by defining both a copy constructor and an equality operator "==" (assume two balloons are equal iff their colours are equal) using your best OO design style.

ANS:

```
Balloon::Balloon(const Balloon& b) : colour(b.colour) {}
bool Balloon::operator==(const Balloon& b) const {
   return b.colour == colour;
}
```

3. **[15 total marks]** *Object-oriented design*

The office of General Accounting of Middle Earth (GAME), has decided to put all creatures on the same payroll system. To start with, they are going to model only Orcs (who are evil) and Elves (who are noble). All creatures have a name. Orcs are paid $100 per murder and $500 per battle they have won; elves are paid $1000 per song they have composed.

All creatures support a method called `printSalary` which causes the following output to be sent to `std::cout` (assuming Oscar has been in 2 battles and committed 4 murders, and that Ernie has composed 2 songs):

```
Oscar the Orc earned $1400
Ernie the Elf earned $2000
```

Your job is to design and implement the three main classes: `Creature`, `Orc`, and `Elf`. Each class should support one constructor (taking the name, and maybe some other data), a destructor, and the `printSalary()` method. You may also add other (non-public) helper methods and variables, if you like.

Assume that this program is going to run only once, at the end of the Middle Earth fiscal year; this means that the creature's stats can be treated as constants, and should be set in the constructor.

Think carefully about how to arrange the various data fields within the hierarchy and how to write the method implementations using the best object-oriented design techniques we have discussed (and maybe a design pattern or two). We will grade heavily on style for this question. Make appropriate use of the `virtual` and `const` keywords. Make methods and variables as hidden as possible. Do *not* use the `typeid` function (if you happen to know what that is) to print the class name.

ANS:

```
class Creature {
   public :
      Creature (const std::string& name, const std::string& kind);
      virtual ˜Creature();
      void printSalary() const;
   private :
      const std::string name;
      const std::string kind;
      virtual int calculateSalary() const = 0;
};

class Orc : public Creature {
   public :
      Orc (const std::string& name, int numBattles, int numMurders);
      virtual ˜Orc();
   private :
      const int numBattles;
      const int numMurders;
      virtual int calculateSalary () const;
};
```

ANS:

```
class Elf : public Creature {
   public :
      Elf (const std::string& name, int numSongs);
      virtual ~Elf();
   private :
      const int numSongs;
      virtual int calculateSalary () const;
};


Creature::Creature(const std::string& name, const std::string& kind)
      : name(name), kind(kind) {}
Creature::~Creature() {}

void Creature::printSalary() const {
   std::cout << name << " the " << kind << " earned $"
      << calculateSalary() << std::endl;
}

Orc::Orc (const std::string& name, int numBattles, int numMurders)
      : Creature (name, "Orc"), numBattles(numBattles),
      numMurders(numMurders) {}
Orc::~Orc(){}

int Orc::calculateSalary() const {
   return 500 * numBattles + 100 * numMurders;
}

Elf::Elf (const std::string& name, int numSongs)
      : Creature (name, "Elf"), numSongs(numSongs) {}
Elf::~Elf(){}

int Elf::calculateSalary() const {
   return 1000 * numSongs;
}
```

4. **[14 marks total — 2 each marks]** *Design patterns*

State the name of the design pattern that best fits each of these situations:

(a) You wish to be able to use several different algorithms for completing the same abstract task. You want to be able to make this decision at run-time.

ANSWER: Strategy

(b) There is an abstraction hierarchy that we wish to manipulate in different ways. The manipulators are tightly coupled (by necessity) to the abstraction hierarchy.

ANSWER: Visitor

(c) It is desired for clients to be able to treat parts and groups of parts uniformly.

ANSWER: Composite

(d) There is a core piece of data that has several active windows that "view" it in some way. When the core data is changed, all of the view objects need to be notified.

ANSWER: Observer

(e) A client knows how to construct an interesting object of several parts, but only abstractly. The client is not related to the parts hierarchy by inheritance.

ANSWER: Builder

(f) A non-virtual public method plus several "pure virtual" private methods are declared in the same class. The public method uses the private methods to abstractly and authoritatively define how something should be done.

ANSWER: Template Method

(g) We wish to decouple an abstraction hierarchy from its various possible implementations, so the two dimensions (abstraction and implementation) can vary independently.

ANSWER: Bridge

5. **[14 total marks]** *Exceptions and exception handling*

(a) **[5 marks]** Write a C++ function `mysqrt` that is a robust version of the C++ standard library function `sqrt`. Your version should take a `double` and return a `double`. If the input is non-negative, it should just return the result from calling `sqrt`. If the input is negative, it should throw a `std::runtime_error` exception, that incorporates an error message of your choosing. Don't worry about which files you need to include, and don't bother to create your own exception class. Just use `std::runtime_error`.

ANS:

```
double mysqrt (double n) throw (std::runtime_error) {
   std::cout << "n = " << n << std::endl;
   if (n < 0) {
      throw std::runtime_error ("Negative passed to mysqrt");
   }
   return sqrt(n);
}
```

(b) **[3 marks]** Consider the following code that calls your function. In the space below, give all of the output that would result (assuming you have implemented `mysqrt` correctly). For simplicity, assume that the square root of 2 is 1.41.

```
try {
    for (int i=2; i>=-2; i--) {
        std::cout << "sqrt = " << mysqrt((double) i) << std::endl;
    }
    std::cout << "All done." << std::endl;
} catch (std::runtime_error& e) {
    std::cout << "Exception: " << e.what() << std::endl;
}
```

ANS: *Note that it's OK if they print a partial line of output for sqrt when an exception is raised.*

sqrt = 1.41
sqrt = 1
sqrt = 0
Exception: Negative value passed to mysqrt

(c) **[3 marks]** Now do the same for this code.

```
for (int i=2; i>=-2; i--) {
    try {
        std::cout << "sqrt = " << mysqrt((double) i) << std::endl;
    } catch (std::runtime_error& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
}
std::cout << "All done." << std::endl;
```

ANS: *Note that it's OK if they print a partial line of output for sqrt when an exception is raised.*

sqrt = 1.41
sqrt = 1
sqrt = 0
Exception: Negative value passed to mysqrt
Exception: Negative value passed to mysqrt
All done.

(d) **[3 marks]** Suppose the function `risky` may throw three different exceptions, all of which are subclasses of `std::runtime_error`. The function `flurble` calls `risky`, and if it detects an exception being raised, it cleans up some local mess and throws the exception again for subsequent handling by its caller. The code below is legal but doesn't quite do the job. Explain what is wrong and how you could fix it.

```
void risky () throw (std::runtime_error) {
    // ...
}

void flurble () throw (std::runtime_error) {
    try {
        risky();  //
    } catch (std::runtime_error e) {
        // ... clean up some local mess (details omitted)
        throw e;
    }
}
```

ANS: *The problem is that the catch creates a new exception object rather than a reference to the existing exception. The two possible fixes are (a) catch a reference instead and (b) just say "throw", which rethrows the original exception.*

6. **[15 total marks]** *STL and C++ programming*

A `QuitQueue` is a queue with the additional facility to allow elements to quit early. The state of a `QuitQueue` can be imagined as a sequence of N entries, which is either empty (when $N = 0$), or contains a sequence of elements T_1, T_2, ..., T_N

The methods on `QuitQueue` are the following, defined in terms of the abstract state:

`empty` does $N = 0$?

`enter (T)` if $N = 0$, then state becomes T, else state becomes T, T_1, T_2, ..., T_N; no value is returned.

`leave` returns T_N and state becomes T_1, T_2, ..., T_{N-1}

`quit(T)` state becomes T_1, ..., T_{i-1}, T_{i+1}, ..., T_N where T_i is the oldest element of the queue that has value T; no value is returned.

`print` print the elements of the QuitQueue, one per line, starting with the oldest element first.

To make life simpler, assume the following:

- the element type is `std::string`, which you may refer to as just plain old `string` in your code.
- `leave`, `quit`, and `print` performed on an empty `QuitQueue` simply do nothing; don't worry about exceptions or error messages.

Your job is to give a full implementation of the `QuitQueue` class. This is not as hard as it sounds if you do a little thinking and use an approrpiate STL container as the "workhorse". Put some thought into which class to use, (think: based on the projected use, which is the most efficient?). You will find an STL "API cheat sheet" as a handout.

(a) **[5 marks]** Give the class declaration of `QuitQueue` with `std::string` as the element type (*i.e.,* don't try to use C++ templates). Don't worry files you might need to include, or `IFNDEF` guards; assume we'll do that for you. Add the `const` modifier where appropriate

ANS:

```
class QuitQueue
   public :
      QuitQueue ();
      ~QuitQueue();
      bool empty () const ;
      void enter (string s);
      string leave ();
      void quit (string s);
      void print() const;
   private :
      std::list<string> q;
;
```

(b) **[10 marks]** Give the implementations of the methods for `QuitQueue`.

ANS:

```
QuitQueue::QuitQueue(){}
QuitQueue::~QuitQueue(){}

bool QuitQueue::empty() const {
   return q.size()==0;
}

void QuitQueue::enter(string s) {
   q.push_back(s);
}

string QuitQueue::leave() {
   if (empty()) {
      return "";
   } else {
      string ans = *q.begin();
      q.pop_front();
      return ans;
   }
}

void QuitQueue::quit(string s) {
   for (std::list<string>::iterator qi=q.begin(); qi!=q.end();
         qi++) {
      if (*qi==s) {
         q.erase(qi);
         break;
      }
   }
}

void QuitQueue::print() const {
   for (std::list<string>::const_iterator qi=q.begin(); qi!=q.end();
         qi++) {
      std::cout << " " << *qi << std::endl;
   }
}
```