Comprehensive Exam: Programming Languages Autumn 2005

1. (*20 points*) ..... Short Answer

   Answer each question in a few words or phrases.

   (a) (*4 points*) In C++, it is possible to designate that a member function `f` is `virtual` in a base class, but not write this designation in the derived class. What relationship between a C++ base class `vtable` and derived class `vtable` implies that the function `f` is also virtual in the derived class? Explain.

   Answer: In C++, it is essential that the `vtable` of a derived class match the `vtable` of the base class. Otherwise, a base class virtual function might not appear in a derived class `vtable`, and C++ subtyping would break.

   (b) (*4 points*) Explain why this program, if compiled and executed without type checking, produces a run-time type error. At which point (and which line of code) will the Java compiler or run-time system report the error?

```
1: class A { ... amethod ...}
2:     class B extends A { ... bmethod ...}
3:     B[ ] bArray = new B[10]
4:     A[ ] aArray = bArray
5:     aArray[0] = new A()
6:     bArray[0].bmethod()
```

   Answer: The program compiles due to Java array covariance. If executed without run-time checking, an error would occur on line 6 because the program attempts to call a class B method on a class A object. In normal Java execution, a run-time exception will be thrown by the assignment `aArray[0] = new A()` in line 5.

   (c) (*4 points*) List three different reasons why Java programs generally run slower than C++ programs.

   Answer: The overhead of interpreting bytecode instructions, array bounds checks, and run-time tests associated with type casts.

   (d) (*4 points*) What is the main advantage of tail recursion optimization? Is it time or space saving? Explain.

   Answer: Space savings. There is some time saving, associated with not having to allocated and deallocate activation records. However, the main consequence of tail recursion optimization is that the amount of space used by a function does not grow in proportion to the number of recursive calls.

   (e) (*4 points*) The designers of C+++ are considering allowing functions to be declared inside any block. As a consequence, a function could be declared inside one function, and passed or returned to another function. What implementation costs are associated with this design option?

   Answer: Functions must be represented by closures, and activation records can no longer be allocated/deallocated in a stack-like (last allocated/first deallocated) manner.

2. (*12 points*) ..... Scope and parameter passing and activation records

Consider this simple program:

```
1   int a :=  1
2   int b := 10
3   proc f(m) {
4      a := a+m
5      b := a+m
6   }
7   proc g(n) {
8      int a := 5
9      f(n)
10  }
11  g(a)
12  print a , b
```

(a) (*6 points*) Assume that all parameters are passed using call-by-value.

i. What values are printed on line 12 under static scoping?
Answer: 2, 3

ii. What values are printed under dynamic scoping?
Answer: 1, 7

(b) (*6 points*) Assume that only static scoping is used.

i. What values are printed if all arguments are passed using call-by-reference?
Answer: 2, 4

ii. What if call-by-value-result is used instead?
Answer: 1, 3

3. (*12 points*) ..... Templates and Generics

(a) (*4 points*) Consider a C++ class of this form:

```
template <class T>
class C<T> {
   ...
int f (T *x, T *y){ ...  x->less(y) ...}
   ...
};
```

Suppose a C++ program contains a an object of type `C<A>` for some C++ class `A`. Explain what functions, operators, and so on, class `A` must define in order for the code shown to compile and link correctly.

Answer: Class `A` must define a member function `less` in such a way that if `A *x` and `A *y` as declared in the argument list of `f`, then `x->less(y)` will type check and compile.

(b) (*4 points*) Suppose that an analogous class `C` is written in Java.

```
class C<T> {
   ...
int f (T x, T y){ ...  x.compareTo(y) ...}
   ...
}
```

Explain why the Java 1.5 compiler would not accept the code, based on the fragments shown.

Answer: This code will not compile because the compiler cannot determine that if `T x` as declared in the code, then `x` has method `compareTo`.

(c) (*4 points*) The Java code can be written so that it will compile-time type check. Instead of `class C<T>`, we will need to write something of the form

```
class C<T extends ...>
```

What would you write in the underlined region to make this correct Java? Explain, and describe any additional classes or interfaces you may need.

```
class C<T extends  ____________________________> {
   ...
int f (T x, T y){ ...  x.compareTo(y) ...}
   ...
}
```

Answer: A partial answer is that we need `T extends Comparable`, where `Comparable` is some class with a `compareTo` operation. However, a full answer should mention the type if `compareTo`. Although this could be a method that is applicable to any object, that doesn't give much of a way to implement interesting comparisons. It's better to define `Comparable` as a parameterized interface `Comparable(T)` with `T` the type of the argument to `compareTo`. Then use `T extends Comparable(T)` in the declaration of `C`.

4. (*16 points*) ..... Concurrency

For the following question, we will use a very simple linked list class designed to be used by multiple threads. The class stores elements in a linked list and whenever an element is added or deleted, the code finds the maximum element in the list and stores it in a member variable. Assume that `max()` will never be called on an empty list.

```
public class CLinkedList
{
  protected CElement head_; //The list's head
  Comparable max_; //A reference to the maximum list element

  public CLinkedList()
  { //Constructor
    head_ = null;
    max_ = null;
  }

  //Assume max() will never be called on an empty list
  public synchronized Comparable max()
  {
    return max_;
  }

  //Add an element to the front of the list
  public synchronized void addElement(Comparable element)
  {
    CElement newHead = new CElement(element, head_);
```

```
  head_ = newHead;
  updateMax(); //Choose a new max element, if necessary
}

//Remove the first element from the list.
public synchronized Comparable deleteFirst()
{
  CElement deletedElement = head_;
  head_ = deletedElement.next; //Remove the first element.
  deletedElement.next = null; //Removed element points nowhere
  updateMax();                //Update the max element
  return deletedElement.element;
}

//Scans each element in the list until it finds an equal element or
//the end of the list.  Returns true if the element is found.
public boolean lookup(Comparable o){
  for (CElement i = head_; i != null; i = i.next){
    if (o.equals(i.element)){
      return true;
    }
  }
  return false;
}

//Scan every element in the list to find the maximum element.  Update
//the max_ member to reference the maximum element.
private void updateMax()
{
  if (head_ != null){
    CElement tempMax = head_;
    for (CElement i = head_; i != null; i = i.next){
      if (tempMax.element.compareTo(i.element) < 0 ||
          tempMax.element.compareTo(i.element) == 0){
        tempMax = i;
      }
    }
    max_ = tempMax.element;
  }
}
//A simple list cell class.
private class CElement
{
  public CElement(Comparable element, CElement next){
    this.element = element;
    this.next = next;
  }
  public Comparable element; //The element referenced
                             //by this list element.
  public CElement next; //The next element in the linked list.
}
```

(a) (*5 points*) A race condition exists in the CLinkedList implementation because `lookup()` is not synchronized. In some cases, a call to `lookup()` may not find an object that is actually in the list. Give a sequence of calls to CLinkedList member functions by different threads that could result in `lookup()` failing to find an object that is in the list. Your answer should include a time-line of method calls made by

specific threads and a description of what causes `lookup()` to fail.

Answer: Yes, we can execute `lookup()`, it finds the first element, and then a new thread is scheduled which deletes the first element and sets the next pointer to null. Therefore, `lookup()` will not search the entire list. Essentially, by not synchronizing `lookup()` it can see the list in an inconsistent state.

Some people may be tempted to say that there is a race condition involving `updateMax()`, but this is not the case. `updateMax()` is a private member function, so we know that it will only be called by `addElement()` and `deleteFirst()`. When called by either of those two functions, the executing thread *retains* its lock on the CLinkedList object because it has not left the scope of the monitor. Therefore, it is not possible for one thread to be executing in `updateMax()` while another is executing and another thread simultaneously executing in either `addElement()` or `deleteFirst()`.

(b) (*4 points*) Can a multi-threaded program become deadlocked when multiple threads concurrently operate on a single CLinkedList object? Explain.

Answer: No, there is only one lock per CLinkedList object – the implicit lock that is associated with every Object in Java. Because of this, no thread can hold a lock while waiting for another lock, and therefore deadlock is not possible.

(c) (*3 points*) If the programmer knows that CLinkedList will only be used on uniprocessor machines, is it necessary for `addElement()` and `deleteFirst()` to be synchronized?

Answer: Yes, because threads can be preempted on uniprocessor machines.

(d) (*4 points*) Calling a synchronized method is more expensive than calling an unsynchronized method because a thread must acquire a lock before running in the monitor. If the programmer knows that `max()` will be called frequently, she would like to optimize the function by making it unsynchronized.

For this problem, keep in mind that the result returned by `max()` may be stale by the time the calling thread uses `max()`'s return value, which is correct. For example, thread 1 could call `max()` and get `max()`'s result, but before using the result thread 2 adds a new element to the list that becomes the new maximum value. The same line of reasoning applies to an unsynchronized version of `max()`.

i. Let's assume that the programmer knows that only `int` elements will be stored in CLinkedLists. Assuming that the programmer can rewrite any necessary code related to the type of elements stored in the linked list, can she make the `max()` method unsynchronized? Explain. For the purposes of this problem, assume that writes to integer variables are atomic, but writes to doubles are not atomic. If an operation is atomic, it either updates any necessary state and completes or it terminates without updating any state at all. If an operation is not atomic, it is possible that it can be interrupted in the middle of its execution and that some intermediate state might be visible to other parts of the program until it is resumed.

Answer: Yes, if max_ is an int, it will be updated atomically, so no matter when max_'s value is read, it will be consistent with some view of the linked list.

ii. If CLinkedList objects will hold only `double` values, can the programmer rewrite code and make `max()` unsynchronized? Explain.

Answer: No, `max()` can not be unsynchronized. Consider a case with two threads on a dual-CPU machine, thread 1 and thread 2. Thread 1 calls `max()` while thread 2 is simultaneously updating max_'s value. Thread 2 writes the first word of max_, thread 1 reads max_, and thread 2 finishes writing the last word of max_. Thread 1 ends up reading the value of max_ in an inconsistent

state, which gives an incorrect result. This situation can occur because writes to doubles are not atomic.