V22-0470-001 Object-Oriented Programming

Final

12/14/10

This exam is a closed book, closed notes exam. It has 3 questions on 19 pages, for a total of 100 points. It also includes an appendix listing the class code for `ptr.h` and `java_lang.h`. Answer the questions in the spaces provided on the question sheets. If you run out of space for an answer, continue on the back of the page.

Name: ____

Question	Points	Score
1	35	
2	35	
3	30	
Total:	100	

1. Hermione is recovering from the Battle of Hogwarts and studying object-oriented programming with Java (go figure). She decides to build a game based on her experiences and identifies the following major entities:

   - A human is a being.
   - A wizard is a human.
   - A house elf is a being.
   - A ghost is a being.

   To complicate matters a little, a ghost also is "the disembodied spirit of a once living wizard or witch" (Harry Potter Wiki).

   She then identifies some state and actions:

   - Every being has a name.
   - Every human has a father and a mother.
   - Every wizard performs magic.
   - Every house elf also performs magic.
   - Every ghost glides to some location.

   When gliding, ghosts tend to pass through physical objects, including walls and other beings — a rather disconcerting experience for those being passed through.

7 (a) How does the "is a" relationship from Hermione's domain analysis translate into well-designed Java code? Please include a specific example.

$\boxed{8}$ (b) How does the fact that wizards and house elves, but not non-wizard humans and ghosts, perform magic translate into well-designed Java code? I.e., how do you express the shared functionality? Please show the key code.

$\boxed{10}$ (c) Turning a wizard into a ghost can be achieved by copying all relevant state from the wizard instance to the newly created ghost instance. What is a more elegant and efficient approach?

$\boxed{10}$ (d) Determining whether a being is, say, a ghost by using `instanceof` is bad programming practice. A better alternative requires adding a new method to `Being` and then overriding that method in `Ghost`. Write both methods.

2. While browsing books in the restricted section of Hogwarts' library, Hermione finds the following Java program:

```
public abstract class Magic {

  public static final Magic STOPPER = new Expelliarmus();
  public static final Magic SLASHER = new Sectumsempra();
  public static final Magic TORTURER = new Crucio();

  public static void show(String s)  { System.out.println(s); }
  public static void show(Magic m)   { show(m.getClass().getSimpleName()); }

  public static Magic cast(Magic m) { return m; }
  public Magic cast() { return this; }
  public Magic counter(Magic m) { return STOPPER; }

  public static class Expelliarmus extends Magic { }

  public static abstract class DarkMagic extends Magic {
    public static Magic cast(Magic m) {
      return m instanceof DarkMagic ? m : SLASHER;
    }
    public Magic counter(Magic m) { return cast(m); }
    public Magic counter(DarkMagic m) { return TORTURER; }
  }

  public static class Sectumsempra extends DarkMagic {
    public Magic cast() { return SLASHER; }
  }

  public static abstract class UnforgivableCurse extends DarkMagic { }

  public static class Crucio extends UnforgivableCurse {
    public Magic counter(Magic m) { return TORTURER; }
  }

  public static void main(String[] args) {
    show(TORTURER.cast(STOPPER));
    show(STOPPER.counter(SLASHER));
    show(SLASHER.counter(SLASHER));
    show(TORTURER.counter(SLASHER));
    show(new Sectumsempra().counter(SLASHER));
  }
}
```

Hermione does remember that `java.lang.Class.getSimpleName()` returns the unqualified name of a class, as written in the class declaration. However, otherwise, she is a little befuddled by this strange little program. Please help her understand it.

5 (a) What is the static type of `SLASHER`?

5 (b) What is the dynamic type of `SLASHER`?

10 (c) What is printed when executing `Magic.main()`?

15 (d) What are the correctly ordered entries in `Sectumsempra`'s vtable for our translator? Please use "*classname*.*methodname*(*typenames*)" notation. Also, ignore the first two entries, which are `__isa` and `__delete`.

3. From the Harry Potter Wiki: "A pair of **Vanishing Cabinets** will act as a passage between two places. Objects placed into one cabinet will appear in the other."

   Hermione decides to implement a C++ template class `VanishingCabinet<T>`, which can transfer one object pointer at a time between two connected cabinets. After creation, a vanishing cabinet is disconnected and empty. In this state, it only signals `bad_magic`. Once connected, a vanishing cabinet cannot be disconnected or reconnected again. An object is placed into a vanishing cabinet by C++ assignment. Placing an object into a cabinet, whose connected cabinet already holds an object, is `bad_magic`. An object is retrieved from a cabinet through a C++ cast, which may be implicit. Retrieving the object also empties that cabinet.

   For example, the code:

```
VanishingCabinet<int> c1, c2;
c1.connect(c2);
cout << c1 << ", " << c2 << endl;

c1 = new int(5);
cout << c1 << ", " << c2 << endl;

c2 = new int(6);
cout << c1 << ", " << c2 << endl;
```

   uses both assignment operators and (implicit) cast operators. It should print something like this:

```
0, 0
0, 0x100100080
0x100100090, 0
```

5 (a) Help Hermione implement `VanishingCabinet` by writing the declaration of its (private) fields.

15 (b) You are on a roll. Keep helping Hermione by writing the assignment operator for `VanishingCabinet`, which places a plain pointer into the connected cabinet. Be careful: This assignment operator does *not* follow any of the established conventions for C++ assignment operators. So look at the example code and carefully think about the performed operation and its types.

10 (c) Write the cast operator `operator T*()` for `VanishingCabinet`, which returns a plain pointer while also emptying the cabinet.

Appendix: ptr.h

```
#include <iostream>
#include <cstring>

#if 0
#define TRACE(s) \
  std::cout << __FUNCTION__ << ":" << __LINE__ << ":" << std::endl
#else
#define TRACE(s)
#endif

namespace __rt {

  template<typename T>
  class Ptr {
    T* addr;
    size_t* counter;

  public:
    typedef T value_t;

    inline Ptr(T* addr = 0) : addr(addr), counter(new size_t(1)) {
      TRACE();
    }

    inline Ptr(const Ptr& other) : addr(other.addr), counter(other.counter) {
      TRACE();
      ++(*counter);
    }

    inline ~Ptr() {
      TRACE();
      if (0 == --(*counter)) {
        if (0 != addr) addr->__vptr->__delete(addr);
        delete counter;
      }
    }

    inline Ptr& operator=(const Ptr& right) {
      TRACE();
      if (addr != right.addr) {
        if (0 == --(*counter)) {
          if (0 != addr) addr->__vptr->__delete(addr);
          delete counter;
        }
        addr = right.addr;
```

```
        counter = right.counter;
        ++(*counter);
      }
      return *this;
    }

    inline T& operator*()  const { TRACE(); return *addr; }
    inline T* operator->() const { TRACE(); return addr;  }
    inline T* raw()        const { TRACE(); return addr;  }

    template<typename U>
    friend class Ptr;

    template<typename U>
    inline Ptr(const Ptr<U>& other)
      : addr((T*)other.addr), counter(other.counter) {
      TRACE();
      ++(*counter);
    }

    template<typename U>
    inline bool operator==(const Ptr<U>& other) const {
      return addr == (T*)other.addr;
    }

    template<typename U>
    inline bool operator!=(const Ptr<U>& other) const {
      return addr != (T*)other.addr;
    }

  };

}
```

Appendix: java_lang.h

```
#pragma once

#include "ptr.h"

#include <stdint.h>
#include <string>

namespace java {
  namespace lang {

    // Forward declarations of data layout and vtables.
    struct __Object;
    struct __Object_VT;

    struct __String;
    struct __String_VT;

    struct __Class;
    struct __Class_VT;

    // Definition of convenient type names.
    typedef __rt::Ptr<__Object> Object;
    typedef __rt::Ptr<__String> String;
    typedef __rt::Ptr<__Class> Class;
  }
}

// ==========================================================================

namespace __rt {

  // The function returning the canonical null value.  See comment
  // below for java::lang::__Object::__class() as to why we use a
  // function.
  java::lang::Object null();

}

// ==========================================================================

namespace java {
  namespace lang {

    // The data layout for java.lang.Object.
    struct __Object {
```

```
  __Object_VT* __vptr;

  __Object()
  : __vptr(&__vtable) {
  };

  // ------------------------------------------------------------------

  // The function returning the class object representing
  // java.lang.Object.
  //
  // We use a function instead of a static field to avoid C++'s
  // "static initialization fiasco".  C++ does not specify the
  // order, in which static fields are initialized.  However,
  // the class object for some type T must point to the class
  // object for its direct superclass S, i.e., the class object
  // for S must be created before the class object for T.
  // The function enforces this ordering by allocating the
  // class object on first invocation and returning the same
  // object on subsequent invocations.
  static Class __class();

  // The virtual destructor.  This method must be virtual
  // because C++'s delete determines the size of the memory
  // to be deallocated based on the pointer's static type.
  // Consequently, every class needs its own __delete(),
  // which simply invokes C++' delete on the pointer.
  static void __delete(__Object*);

  // The methods implemented by java.lang.Object.
  static int32_t hashCode(Object);
  static bool equals(Object, Object);
  static Class getClass(Object);
  static String toString(Object);

  // The vtable for java.lang.Object.
  static __Object_VT __vtable;
};

// The vtable layout for java.lang.Object.
struct __Object_VT {
  Class __isa;
  void (*__delete)(__Object*);
  int32_t (*hashCode)(Object);
  bool (*equals)(Object, Object);
  Class (*getClass)(Object);
```

```
  String (*toString)(Object);

  __Object_VT()
  : __isa(__Object::__class()),
    __delete(&__Object::__delete),
    hashCode(&__Object::hashCode),
    equals(&__Object::equals),
    getClass(&__Object::getClass),
    toString(&__Object::toString) {
  }
};

// ==========================================================================

// The data layout for java.lang.String.
struct __String {
  __String_VT* __vptr;
  std::string data;

  __String(std::string data)
  : __vptr(&__vtable),
    data(data) {
  }

  // The function retturning the class object representing
  // java.lang.String.
  static Class __class();

  // The virtual destructor.
  static void __delete(__String*);

  // The methods implemented by java.lang.String.
  static int32_t hashCode(String);
  static bool equals(String, Object);
  static String toString(String);
  static int32_t length(String);
  static char charAt(String, int32_t);

  // The vtable for java.lang.String.
  static __String_VT __vtable;
};

// The vtable layout for java.lang.String.
struct __String_VT {
  Class __isa;
  void (*__delete)(__String*);
```

```
  int32_t (*hashCode)(String);
  bool (*equals)(String, Object);
  Class (*getClass)(String);
  String (*toString)(String);
  int32_t (*length)(String);
  char (*charAt)(String, int32_t);

  __String_VT()
  : __isa(__String::__class()),
    __delete(&__String::__delete),
    hashCode(&__String::hashCode),
    equals(&__String::equals),
    getClass((Class(*)(String))&__Object::getClass),
    toString(&__String::toString),
    length(&__String::length),
    charAt(&__String::charAt) {
  }
};

// The overloaded output operator for java.lang.String.
inline std::ostream& operator<<(std::ostream& out, String s) {
  return out << s->data;
}

// =======================================================================

// The data layout for java.lang.Class.
struct __Class {
  __Class_VT* __vptr;
  String name;
  Class parent;
  Class component;
  bool primitive;

  __Class(String name, Class parent,
          Class comp = __rt::null(), bool prim = false)
  : __vptr(&__vtable),
    name(name),
    parent(parent),
    component(comp),
    primitive(prim) {
  }

  // ---------------------------------------------------------------------

  // The function returning the class object representing
```

```
  // java.lang.Class.
  static Class __class();

  // The virtual destructor.
  static void __delete(__Class*);

  // The instance methods of java.lang.Class.
  static String toString(Class);
  static String getName(Class);
  static Class getSuperclass(Class);
  static Class getComponentType(Class);
  static bool isPrimitive(Class);
  static bool isArray(Class);
  static bool isInstance(Class, Object);

  // The vtable for java.lang.Class.
  static __Class_VT __vtable;
};

// The vtable layout for java.lang.Class.
struct __Class_VT {
  Class __isa;
  void (*__delete)(__Class*);
  int32_t (*hashCode)(Class);
  bool (*equals)(Class, Object);
  Class (*getClass)(Class);
  String (*toString)(Class);
  String (*getName)(Class);
  Class (*getSuperclass)(Class);
  bool (*isPrimitive)(Class);
  bool (*isArray)(Class);
  Class (*getComponentType)(Class);
  bool (*isInstance)(Class, Object);

  __Class_VT()
  : __isa(__Class::__class()),
    __delete(__Class::__delete),
    hashCode((int32_t(*)(Class))&__Object::hashCode),
    equals((bool(*)(Class,Object))&__Object::equals),
    getClass((Class(*)(Class))&__Object::getClass),
    toString(&__Class::toString),
    getName(&__Class::getName),
    getSuperclass(&__Class::getSuperclass),
    isPrimitive(&__Class::isPrimitive),
    isArray(&__Class::isArray),
    getComponentType(&__Class::getComponentType),
```

```
    isInstance(&__Class::isInstance) {
  }
};

// ======================================================================

// The completely incomplete data layout for java.lang.Integer.
struct __Integer {
  // Instance fields would go here.

  // The function returning the class object representing
  // primitive ints.
  static Class __primitiveClass();
};

// ======================================================================

// For simplicity, we use C++ inheritance for exception types
// and throw them by value (see below).  In other words, the
// translator does not support user defined exceptions and simply
// uses a few built-in classes.
class Throwable {
};

class Exception : public Throwable {
};

class RuntimeException : public Exception {
};

class NullPointerException : public RuntimeException {
};

class ArrayStoreException : public RuntimeException {
};

class ClassCastException : public RuntimeException {
};

class IndexOutOfBoundsException : public RuntimeException {
};

class ArrayIndexOutOfBoundsException : public IndexOutOfBoundsException {
};

// ======================================================================
```

```
// Forward declarations of data layout and vtables.
template <typename T>
struct __Array;

template <typename T>
struct __Array_VT;

// Definition of convenient type names.
typedef __rt::Ptr<__Array<int32_t> > ArrayOfInt;
typedef __rt::Ptr<__Array<Object> > ArrayOfObject;
typedef __rt::Ptr<__Array<Class> > ArrayOfClass;

// The data layout for arrays.
template <typename T>
struct __Array {
  __Array_VT<T>* __vptr;
  const int32_t length;
  T* __data;

  __Array(const int32_t length)
  : __vptr(&__vtable), length(length), __data(new T[length]) {
    // Only zero out __data for arrays of primitive types.
  }

  static Class __class();

  static void __delete(__Array* __this) {
    delete[] __this->__data;
    delete __this;
  }

  T& operator[](int idx) {
    if (0 > idx || idx >= length) throw ArrayIndexOutOfBoundsException();
    return __data[idx];
  }

  const T& operator[](int idx) const {
    if (0 > idx || idx >= length) throw ArrayIndexOutOfBoundsException();
    return __data[idx];
  }

  static __Array_VT<T> __vtable;
};

// The vtable layout for arrays.
```

```
    template <typename T>
    struct __Array_VT {
      typedef __rt::Ptr<__Array<T> > Array;

      Class __isa;
      void (*__delete)(__Array<T>*);
      int32_t (*hashCode)(Array);
      bool (*equals)(Array, Object);
      Class (*getClass)(Array);
      String (*toString)(Array);

      __Array_VT()
      : __isa(__Array<T>::__class()),
        __delete(&__Array<T>::__delete),
        hashCode((int32_t(*)(Array))&__Object::hashCode),
        equals((bool(*)(Array,Object))&__Object::equals),
        getClass((Class(*)(Array))&__Object::getClass),
        toString((String(*)(Array))&__Object::toString) {
      }

    };

    // The header file declares each template (see above) just as for
    // regular C++ classes.  However, since the compiler needs to know
    // how to instantiate each template, the header file also defines
    // each template.

    // The vtable for arrays.  Note that this definition uses the
    // the default no-arg constructor.
    template <typename T>
    __Array_VT<T> __Array<T>::__vtable;

    // But where is the definition of __class()???

  }
}

// ==========================================================================

namespace __rt {

  // Convenience function for converting C++ strings (std::string)
  // into translated Java strings (java::lang::String).
  // The C++ compiler automaticcally converts C string literals
  // (const char *) into C++ stirngs, so we only need one such function.
  inline java::lang::String stringify(std::string s) {
```

```
    return new java::lang::__String(s);
  }

  // Template function to check against null values.
  template<typename T>
  void checkNotNull(T o) {
    if (null() == o) throw java::lang::NullPointerException();
  }

  // Template function to check array stores.
  template<typename T, typename U>
  void checkArrayStore(Ptr<java::lang::__Array<T> > array, U object) {
    if (null() != object) {
      java::lang::Class arraytype = array->__vptr->getClass(array);
      java::lang::Class eltype = arraytype->__vptr->getComponentType(arraytype);

      if (! eltype->__vptr->isInstance(eltype, (java::lang::Object)object)) {
        throw java::lang::ArrayStoreException();
      }
    }
  }

  // Template function to perform Java casts.
  template<typename Target, typename Source>
  Target java_cast(Source other) {
    java::lang::Class k = Target::value_t::__class();

    if (! k->__vptr->isInstance(k, other)) {
      throw java::lang::ClassCastException();
    }

    return Target(other);
  }

}
```