Homework 7

Due 26 November

Handout 21
CS242: Autumn 2003
19 November

____
(Print your name)

Prob	# 1	# 2	# 3	# 4	# 5	# 6	# 7	8	Total
C+									
C									
C-									

Reading

1. Chapter 12, C++, and Chapter 13, Java.

Problems

1. .... C++ Conversions

C++ allows the programmer to specify how to convert between different classes. This can be very useful when you are passing around objects by value, as the default behavior is not always what is desired. If you are unfamiliar with C++, you may find Bruce Eckel's *Thinking in C++, 2nd Ed* helpful. It is freely available on the web, via http://www.mindview.net/Books/.

(a) What is printed by the following code and why? Your answer should be three sentences or less.

```
#include <iostream>
using namespace std;

class A {
public:
virtual void foo() { cout << ``A::foo\n'' << endl; }
};

class B : public A {
public:
virtual void foo() { cout << ``B::foo\n'' << endl; }
};

void test( A a );

void test( A a ) {
a.foo();
};

int main() {
A a;
B b;
```

```
test(a);
test(b);
}
```

(b) C++ provides an interesting mix of features for converting between objects of different types. The two primary language features are copy constructors and operator conversion. Like many parts of C++ they can surpise people. Explain why the following code prints

```
bork
bork
```

instead of

```
bork
B::foo
```

or

```
bork
bark
```

could this problem be solved by using a copy constructor instead? Explain briefly.

```
#include <iostream>
using namespace std;

class A {
public:
        char *s;
        virtual void foo() { cout << s << endl; }
        A() : s(``bork'') {}
};

class B : public A {
public:
        virtual void foo() { cout << ``B::foo'' << endl; }
        operator A() const { A a; a.s = ``bark''; return a; }
};

void test( A a );

void test( A a ) {
        a.foo();
};

int main() {
        A a;
        B b;
        test(a);
        test(b);
}
```

(c) C++ is the only mainstream object oriented language that does not hide its objects behind pointers or references. You have just seen how this can cause an intersting set of problems when programmers pass objects by value. What two character edit can you do to the original code to make it work as expected?

2. .... C++ Multiple Inheritance and Casts

An important aspect of C++ object and virtual function table (`vtbl`) layout is that if class `D` has class `B` as a public base class, then the initial segment of every `D` object must look like a `B` object, and similarly for the `D` and `B` virtual function tables. The reason is that this makes it possible to access any `B` member data or member function of a `D` object in exactly the same way we would access the `B` member data or member function of a `B` object. While this works out fairly easily with only single inheritance, some effort must be put into the implementation of multiple inheritance to make access to member data and member functions uniform across publicly derived classes.

Suppose class `C` is defined by inheriting from classes `A` and `B`:

```
class A {
    public:
       int x;
       virtual void f();
};
class B {
    public:
       int y;
       virtual void f();
       virtual void g();
};
class C : public A, public B {
    public:
       int z;
       virtual void f();
```

```
};
C *pc = new C;    B *pb = pc;     A *pa = pc;
```

and pa, pb and pc are pointers to the same object, but with different types. The representation of this object of class C and the values of the associated pointers are illustrated in this chapter.

(a) Explain the steps involved in finding the address of the function code in the call `pc->f()`. Be sure to distinguish what happens at compile time from what happens at run time. Which address is found, `&A::f()`, `&B::f()`, or `&C::f()`?

(b) The steps used to find the function address for `pa->f()` and to then call it are the same as for `pc->f()`. Briefly explain why.

(c) Do you think the steps used to find the function address for and to call `pb->f()` have to be the same as the other two, even though the offset is different? Why or why not?

(d) How could the call `pc->g()` be implemented?

3. .... Multiple Inheritance and Thunks

Suppose class C is defined by inheriting from classes A and B:

```
class A {
    public:
       virtual void g();
       int x;
};
class B {
    public:
       int y;
       virtual B* f();
       virtual void g();
};
class C : public A, public B {
    public:
       int z;
       virtual C* f();
       virtual void g();
};
C *pc = new C;   B *pb = pc;     A *pa = pc;
```

and pa, pb and pc are pointers to the same object, but with different types.

Then, pa and pc will contain the same value, but pb will contain a different value; it will contain the address of the B part of the C object. The fact that pb and pc do not contain the same value means that in C++, a cast sometimes has a run-time cost. (In C this is never the case.)

Note that in our example `B::f` and `C::f` do not return the same type. Instead, `C::f` returns a `C*` while `B::f` returns a `B*`. That is legal because C is derived from B, and thus a `C*` can always be used in place of a `B*`.

However, there is a problem: when the compiler sees `pb->f()` it doesn't know whether the call will return a `B*` or a `C*`. Since the caller is expecting a `B*`, the compiler must make sure to return a valid pointer to a `B*`. The solution is to have the C-as-B vtable contain a pointer to a *thunk*. The thunk calls `C::f`, and then adjusts the return value to be a `B*`, before returning.

(a) Draw all of the vtables for the classes in these examples. Show to which function each vtable slot points.

(b) Does the fact that casts in C++ sometimes have a run-time cost, while C never does, indicate that C++ has not adhered to the principles given in class for its design? Why or why not?

(c) Since thunks are expensive, and since C++ only charges programmers for features they use, there must be a feature, or combination of features, that are imposing this cost. What feature or features are these?

4. .... Dispatch on State

One criticism of dynamic dispatch as found in C++ and Java is that it is not flexible enough. The operations performed by methods of a class usually depend on the state of the receiver object. For example, we have all seen code similar to the following file implementation:

```
class StdFile {
   private:
      enum { OPEN, CLOSED } state; /* state can only be either OPEN or CLOSED */

   public:
      StdFile() { state = CLOSED; }           /* initial state is closed */
      void Open() {
         if (state == CLOSED) {
            /* open file ...  */
            state = OPEN;
         } else {
            error "file already open";
         }
      }
      void Close() {
         if (state == OPEN ) {
            /* close file ...  */
            state = CLOSED;
         } else {
            error "file not open";
         }
      }
   }
}
```

Each method must determine the state of the object (*i.e.,* whether or not the file is already open) before performing any operations. Because of this, it seems useful to extend dynamic dispatch to include a way of dispatching not only on the class of the receiver, but also on the state of the receiver. Several object-oriented programming languages, including BETA and Cecil, have various mechanisms to do this. This problem will examine two ways in which we can extend dynamic dispatch in C++ to depend on state. First, we present dispatch on three pieces of information:

- the name of the method being invoked
- the type of the receiver object

- the explicit `state` of the receiver object

As an example, the following declares and creates objects of the `File` class using the new dispatch mechanism

```
class File {
   state in { OPEN, CLOSED };     /* declare states that a File object may be in */

   public:
      File() { state = CLOSED; }     /* initial state is closed */
      switch(state) {

       case CLOSED: {
         void Open() {             /* 1 */
            /* open file ...  */
            state = OPEN;
         }
         void Close() {
            error "file not open";
         }
      }

       case OPEN: {
         void Open() {            /* 2 */
            error "file already open";
         }
         void Close() {
            /* close file ...  */
            state = CLOSED;
         }
      }

}

File* f = new File();
f->Open();       /* calls version 1 */
f->Open();       /* calls version 2 */
...
```

The idea is that the programmer can provide a different implementation of the same method for each state that the object can be in.

(a) Describe one advantage of having this new feature, *i.e.*, are there any advantages to writing classes like `File` over classes like `StdFile`. Describe one disadvantage of having this new feature.

(b) For this part of the problem, assume that subclasses can not add any new states to the set of states inherited from the base class. Describe an object representation that allows for efficient method lookup. Method call should be as fast as virtual method calls in C++, and changing the state of an object should be a constant time operation. (Hint: you may want to have a different vtable for each state). Is this implementation acceptable according to the C++ design goal of only paying for the features which you use?

(c) What problems arise if subclasses are allowed to extend the set of possible states? For example, we could now write a class like:

```
class SharedFile:  public File {
   state in { OPEN, CLOSED, READONLY };     /* extend the set of states */
   ...
}
```

Do not try to solve any of these problems. Just identify several of them.

(d) We may generalize this notion of dispatch base on the state of an object to dispatch based on any predicate test. For example, consider the following Stack class:

```
class Stack {
```

```
    private:
       int n;
       int elems[100];

    public:
       Stack() { n = 0; }

       when(n == 0) {
          int Pop() {
             error "empty";
          }
       }

       when(n > 0) {
          int Pop() {
             return elems[--n];
          }
       }
       ...
}
```

Is there an easy way to extend your proposed implementation in part b to handle dispatch on predicate tests? Why or why not?

5. .... Java `final` and `finalize`

Java has keywords `final` and `finalize`.

(a) Describe one situation where you would want to mark a class `final`, and another where you would want a final method but not a final class.

(b) Describe the similarity and differences between Java `final` and the C++ use of non-`virtual` in similar situations.

(c) Why is Java `finalize` (the other keyword!) a useful feature?

6. .... Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. In contrast, a Java derived class may only have one base class but may implement more than one interface. This question asks you to compare these two language designs.

(a) Draw a C++ class hierarchy with multiple inheritance using the following classes:

Pizza, for a class containing all kinds of pizza,
Meat, for pizza that has meat topping,
Vet, for pizza that has vegetable topping,
Sausage, for pizza that has sausage topping,
Ham, for pizza that has ham topping,
Pineapple, for pizza that has pineapple topping,
Mushroom, for pizza that has mushroom topping,
Hawaiian, for pizza that has ham and pineapple topping.

For simplicity, treat sausage and ham as meats and pineapple and mushroom as vegetables.

(b) If you were to implement these classes in C++, for some kind of pizza manufacturing robot, what kind of potential conflicts associated with multiple inheritance might you have to resolve?

(c) If you were to represent this hierarchy in Java, which would you define as interfaces and which as classes? Write your answer by carefully redrawing your picture, identifying which are classes and which are interfaces. If your program creates objects of each type, you may need to add some additional classes. Include these in your drawing.

(d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.

7. .... Adding pointers to Java

Java does not have general pointer types. More specifically, Java has primitive types (Booleans, integers, floating point numbers, ...) and reference types (objects and arrays). If a variable has a primitive type, then the variable is associated with a location and a value of that type is stored in that location. When a variable of primitive type is assigned a new value, a new value is copied into the location. In contrast, variables of reference type are implemented as pointers. When a variable referring to an object is assigned a value, the location associated with the variable will contain a pointer to the appropriate object.

Imagine that you were part of the Java design team and you believe strongly in pointers. You want to add pointers to Java, so that for every type `A`, there is a type `A*` of pointers to values of type `A`. Gosling is strongly opposed to adding an "address of" operator (like & in C), but you think there is a useful way of adding pointers without adding address-of.

One way of designing a pointer type for Java is to consider `A*` equivalent to the following class:

```
class A* {
    private A data=null;
    public void assign(A x) {data=x;};
    public A deref(){return data;}
    A*(){};
};
```

Intuitively, a pointer is an object with two methods, one assigning a value to the pointer and the other dereferencing a pointer to get the object it points to. One pointer, `p`, can be assigned the object reached by another, `q`, by writing `p.assign(q.deref())` The constructor `A*` does not do anything because the initialization clause sets every new pointer using the null reference.

(a) If `A` is a reference type, do `A*` objects seem like pointers to you? More specifically, suppose `A` is a Java class with method `m` that has a side effect on the object and consider the following code:

```
A x = new A(...);
A* p = new A*();
p.assign(x);
(p.deref()).m();
```

Here, pointer `p` points to the object named by `x` and `p` is used to invoke a method. Does this modify the object named by `x`? Answer in one or two sentences.

(b) What if `A` is a primitive type, such as `int`? Do `A*` objects seem like pointers to you? (*Hint:* Think about the code in part (a).) Answer in one or two sentences.

(c) If $A <: B$, should $A* <: B*$? Answer this question by comparing the type of `A*::assign` to the type of `B*::assign` and comparing the type of `A*::deref` to the type of `B*::deref`.

(d) If Java had templates, then you could define a pointer template `Ptr`, with $\mathtt{Ptr}\langle\mathtt{A}\rangle$ defined like `A*` above. One of the issues that arises in adding templates to Java is subtyping for templates. Based on the `Ptr` example, do you think it is correct to assume that for every class template `Template`, if $A <: B$ then $\mathtt{Template}\langle\mathtt{A}\rangle <: \mathtt{Template}\langle\mathtt{B}\rangle$? Explain briefly.

8. .... Java Generics

One important language feature that is missing from Java is parametric polymorphism (generics), which allows the same piece of code (e.g. a stack implementation) to operate on different types of objects. The common workaround is to use the universal superclass `Object` when objects of different types are expected. However, people often find it to be inconvenient, error prone, and inefficient compared to a generic implementation (e.g. C++ Standard Template Library).

There are two general ways to implement parameterized classes (templates). We will call these the homogeneous and the heterogeneous approaches. In a homogeneous implementation, a type parameter in a generic construct is given some specific type, such as `Object`, and run-time casts are used as needed to provide appropriate functionality for different actual types at run time. The "type erasure" implementation for Java 1.5 is an example of a homogeneous implementation*.

In a heterogeneous implementation, different uses of a generic construct may be implemented differently at run time. One way to do this is simply to compile each use of a template separately. Another implementation strategy produces a generic implementation of a template at

*Interested (or sufficiently confused) readers might want to refer to examples (on pages 4 and 6) in the following paper: `http://www.cis.unisa.edu.au/%7Epizza/gj/Documents/gj-oopsla.pdf`.

compile time, then instantiates the generic implementation before the code is executed. The C++ implementation of templates is a hetergeneous implementation. In Java, separately compiled templates could be instantiated at load time or at run time.

In thinking about implementing templates for Java, keep in mind that the virtual machine was designed and built for a version of Java without templates.

(a) Describe *briefly* why the workaround described in the first paragraph is often **inconvenient**, **error prone** and **inefficient**. Ideally, your answer should contain only one sentence for each of the three shortcomings.

(b) Which approach, homogeneous or heterogeneous, will give better backward compatibility with existing compiled Java code? (Hint: consider the case where you need to pass a generic linked list to some legacy code that expects a list of `Objects`).

(c) What are the performance (timing) trade-offs between homogeneous and heterogeneous implementations? Consider both compile-load-link time costs and run-time costs. (Hint: recall the discussion on the implementation of parametric polymorphism in ML and C++ in Section 6.4.2 of the book.)

(d) What are the space trade-offs between homogeneous and heterogeneous implementations? Consider both space used to represent data and code space. (Hint: again, Section 6.4.2

should help).

(e) Which approach allows more flexibility for supporting language features? Explain. Hint: think about how or whether you can translate templates that create new objects of the parameter type, as in this code fragment:

```
class MyClass <T> {
       ...
       T myobject = new T();
... }
```