Homework 6

Due 17 November

Handout 7
CS242: Autumn 2004
10 November

Reading

1. Finish reading Chapter 11 on Simula and Smalltalk.
2. Read Chapter 12 on C++.

Problems

1. .... Smalltalk Implementation Decisions

In Smalltalk, each class contains a pointer to the class template. This template stores the names of all the instance variables that belong to objects created by the class.

(a) The names of the methods are stored next to the method pointers. Why are the names of instance variables stored in the class, instead of in the objects (next to the values for the instance variables)?

(b) Each class's method dictionary only stores the names of the methods explicitly written for that class; inherited methods are found by searching up the superclass pointers at run-time. What optimization could be done if each subclass contained all of the methods of its superclasses in its method dictionary? What are some of the advantages and disadvantages of this optimization?

(c) The class template stores the names of all the instance variables, even those inherited from parent classes. These are used to compile the methods of the class. Specifically, when a subclass is added to a Smalltalk system, the methods of the new class are compiled so that when an instance variable is accessed, the method can access this directly without searching through the method dictionary and without searching through superclasses. Can you see some advantages and disadvantages of this implementation decision, in comparison with looking up the relative position of an instance variable in the appropriate class template each time the variable is accessed? Keep in mind that in Smalltalk, a set of classes could remain running for days, while new classes are added incrementally and, conceivably, existing classes could be rewritten and recompiled.

2. .... Protocol Conformance

We can compare Smalltalk interfaces to classes using *protocols,* which are lists of operation names (selectors). When a selector allows parameters, as in `at:    put:` , the selector name includes the colons but not the spaces. More specifically, if `dict` is an updatable collection object, such as a dictionary, then we could send `dict` a message by writing `dict at:'cross' put:'angry'.` (This makes our dictionary definition of "cross" the single word "angry.") The protocol for updatable collections will therefore contain the seven-character selector name `at:put:`. Here are some example protocols.

```
             stack : {isEmpty, push:, pop }
             queue : {isEmpty, insert:, remove }
    priority_queue : {isEmpty, insert:, remove }
           dequeue : {isEmpty, insert:,insertFront:, remove, removeLast }
 simple_collection : {isEmpty }
```

Briefly, a stack can be sent the message `isEmpty`, returning `true` if empty, `false` otherwise; `push:` requires an argument (the object to be pushed onto the stack), `pop` removes the top element from the stack and returns it. Queues work similarly, except they are first-in/first-out instead of first-in/last-out. Priority queues are first-in/minimum-out and dequeues are doubly-ended queues with the possibility of adding and removing from either end. The `simple_collection` class just collects methods that are common to all the other classes. We say that the protocol for `A` *conforms* to the protocol for `B` if the set of `A` selector names contains the set of `B` selector names.

(a) Draw a diagram of these classes, ordered by protocol conformance. You should end up with a graph that look's like William Cook's drawing shown in the text.

(b) Describe briefly, in words, a way of implementing each class so that you only make `B` a subclass of `A` if the protocol for `B` conforms to the protocol set for `A`.

(c) For some classes `A` and `B` that are unrelated in the graph, describe a strategy for implementing `A` as a subclass of `B` in a way that keeps them unrelated.

(d) Describe implementation strategies for two classes `A` and `B` (from the set of classes above) so that `B` is a subclass of `A`, but `A` conforms to `B`, not the other way around.

3. .... Removing a Method

Smalltalk has a mechanism for "undefining" a method. Specifically, if a class `A` has method `m` then a programmer may cancel `m` in subclass `B` by writing

```
m:
   self shouldNotImplement
```

With this declaration of `m` in subclass `B`, any invocation of `m` on a `B` object will result in a special error indicating that the method should not be used.

(a) What effect does this feature of Smalltalk have on the relationship between inheritance and subtyping?

(b) Suppose class `A` has methods `m` and `n`, and method `m` is canceled in subclass `B`. Method `n` is inherited and not changed, but method `n` sends the message `m` to `self`. What do you think happens if a `B` object `b` is sent a message `n`? There are two possible outcomes. See if you can identify both, and explain which one you think the designers of Smalltalk would have chosen and why.

4. .... Assignment and Derived Classes

This problem exams the difference between two forms of object assignment. In C++, local variables are stored on the run-time stack, while dynamically allocated data (created using the `new` keyword) is stored on the heap. A local object variable allocated on the stack has an object L-value and an R-value. Unlike many other object-oriented languages, C++ allows object assignment into object L-values.

A C++ programmer writes the following code:

```
class Vehicle {
public:
    int x;
    virtual void f();
    void g();
};

class Airplane :  public Vehicle {
public:
    int y;
    virtual void f();
    virtual void h();
};

void inHeap() {
    Vehicle *b1 = new Vehicle;       // Allocate object on the heap
    Airplane *d1 = new Airplane;     // Allocate object on the heap
    b1->x = 1;
    d1->x = 2;
    d1->y = 3;
    b1 = d1;                         // Assign derived class object to base class pointer
}

void onStack() {
    Vehicle b2;                      // Local object on the stack
    Airplane d2;                     // Local object on the stack
    b2.x = 4;
    d2.x = 5;
    d2.y = 6;
    b2 = d2;                         // Assign derived class object to base class variable
}

int main() {
    inHeap();
    onStack();
}
```

(a) Draw a picture of the stack, heap and vtables that result after objects `b1` and `d1` have been allocated (but *before* the assignment `b1=d1`) during the call to `inHeap`. Be sure to indicate where the instance variables and vtable pointers of the two objects are stored before the assignment `b1=d1`, and to which vtables the respective vtable pointers point.

(b) Re-draw your diagram from (a), showing the changes that result after the assignment `b1=d1`. Be sure to clearly indicate where `b1`'s vtable pointer points after the assignment `b1=d1`.
Explain why `b1`'s vtable pointer points where it does after the assignment `b1=d1`.

(c) Draw a picture of the stack, heap and vtables that result after objects `b2` and `d2` have been allocated (but *before* the assignment `b2=d2`) during the call to `onStack`. Be sure to indicate where the instance variables and vtable pointers of the two objects are stored before the assignment `b2=d2`, and to which vtables the respective vtable pointers point.

(d) In C++, assignment to objects (such as `b2=d2`) is performed by copying into all the left object's data members (`b2`'s in our example) from the right object's (`d2`'s) corresponding data members. If the right object contains data members not present in the left object, then those data simply aren't copied. In `b2=d2` this means overwriting all b2's data members with the corresponding value from d2.

   i. Re-draw your diagram from (c), showing the changes that result after the assignment `b2=d2`.
   ii. Explain why it isn't sensible to copy all of d2's member data into b2's record.
   iii. Explain why b2's vtable pointer isn't changed by the assignment.

(e) We have used assignment statements `b1=d1` and `b2=d2`. Why are the opposite statements `d1=b1` and `d2=b2` not allowed?

5. .... Subtyping and Public Data

This question asks you to review the issue of specializing the types of public member functions and consider the related issue of specializing the types of public data members. To make this question concrete, we will use the example of circles and colored circles in C++. Colored circles get darker each time they are moved. We assume there is a class `Point` of points and class `ColPoint` of colored points with appropriate operations. Class `Point` is a public base class of `ColPoint`. The basic definition for circle is:

```
class Circle {
public:
   Circle(Point* c, float r) { center=c; radius=r; }

   Point* center;
   float radius;
   virtual Circle* move(float dx, float dy)
               {center->move(dx,dy); return(this);}
};
```

For each of the following definitions of colored circle, explain why a colored circle should or should not be considered a subtype of a circle, in principle. If a colored circle should not be a subtype, give some fragment of code that would be type correct if we considered colored circles to be a subtype of circles, but would lead to a type error at run time.

(a)
```
class ColCircle : public Circle {
public:
   ColCircle(Point* c, float r, color cl) : Circle(c,r) { col=cl; }

   color col;
   virtual Circle* move(float dx, float dy)
               {center->move(dx,dy); darken(); return(this);}
   virtual void darken()
               {if (col==green) col=darkgreen;}
};
```

(b) Suppose we change the definition of `ColCircle` so that `move` returns a colored circle.

```
class ColCircle : public Circle {
public:
   ColCircle(Point* c, float r, color cl) : Circle(c,r) { col=cl; }

   color col;
   virtual ColCircle* move(float dx, float dy)
               {center->move(dx,dy); darken(); return(this);}
   virtual void darken()
               {if (col==green) col=darkgreen;}
};
```

(c) Suppose that instead of representing the color of a circle by a separate data member, we replace points by colored points and use a colored point for the center of the circle. One advantage of doing so is that we can use all of the color operations provided for colored points, as illustrated by the darken member function below. This could be useful if we wanted to have the same color operations on a variety of geometric shapes.

```
class ColCircle : public Circle {
public:
   ColCircle(ColPoint* c, float r) : Circle(c,r) { }

   ColPoint* center;

   virtual ColCircle* move(float dx, float dy)
```

```
                 {center->move(dx,dy); darken(); return(this);}
   virtual void darken()
                 {center->darken();}
};
```

The important issue is the redefinition of the type of center. Assume that each `ColCircle` object has `center` and `radius` data fields at the same offset as the `center` and `radius` fields of a `Circle` object. In the modified version of C++ considered in this question, the declaration `ColPoint* center` does *not* mean that a `ColCircle` object has two `center` data fields.

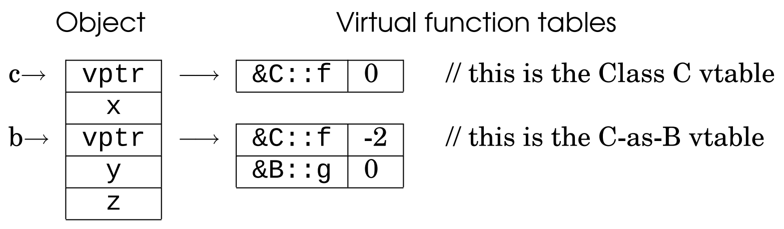
6. .... Multiple Inheritance and Thunks

Most contemporary implementations of C++ use a form of "thunk" in the implementation of multiple inheritance, instead of the offset δ's described in the book. Each thunk encodes the adjustment to the `this` pointer that is stored in the vtable in the original implementation. Although some of the mechanics may seem a little more complicated, the thunk implementation technique has some definite advantages.

Consider class `C` defined by inheriting from classes `A` and `B`, and the following call to a virtual function:

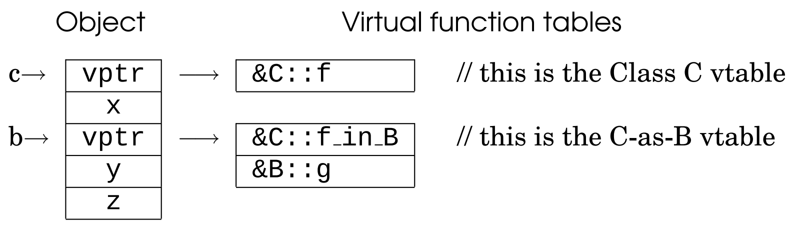
```
class A { int x; virtual void f(); }
class B { int y; virtual void f();
                 virtual void g(); }
class C : A, B { int z; void f(); }

C *c = new C;
B *b = c;
b->f(); // Calls C::f()
```

Using the implementation of multiple inheritance described in the book, the run-time structures associated with this code have the following form:



In this illustration, the first entry in each row of the vtable is the address of the function that will be called, and the second entry of the row is an offset that needs to be added to the `this` pointer as part of the calling sequence.

In the alternate thunk-based implementation, the run-time structures look like this:



In this implementation, `C::f_in_B` is a thunk that subtracts 2 from the `this` pointer and then calls `C::f`. Here is sample code for `C::f_in_B`, using a `goto` to avoid the usual calling sequence and creation of an activation record:

```
C::f_in_B(void *this){
    this = this - 2;
    goto C::f;
}
```

This problem asks you to compare the two implementations.

(a) If a C++ program uses only single inheritance, and no multiple inheritance, are offsets (in implementation 1) or thunks (in implementation 2) needed? Explain?

(b) Suppose you built a compiler for C++ programs using only single inheritance, then modified it, using implementation 1, for multiple inheritance. How does the size of each vtable change? More specifically, suppose that you compile code defining and using a class. If the vtable for this class has size n under single inheritance, what size will it have when compiled using your modified (multiple inheritance) compiler?

(c) Under the same conditions (modifying a single-inheritance compiler to support multiple inheritance, using the offset-based first implementation), how does the execution time required to call a virtual function change?

(d) Now consider the questions posed in parts (b) and (c) for the second, thunk-based, implementation. How does the size of a the vtable for a base class (a class that is not derived from any class) change? How does the execution time required to call a virtual function (for a base-class object) change?

(e) What kind of programs run more efficiently using the first implementation than the second, and what kind of programs run more efficiently using the second implementation than the first? Your answer will probably include single inheritance, multiple inheritance, and the what that objects are frequently manipulated in the program. You do not need to write more than two or three sentences.

(f) What basic principle in the design of C++ seems to be more closely followed in the second (thunk-based) implementation than the first (the offset-based implementation)?