Homework 7

Due 26 November

Reading

1. Chapter 12, C++.

Problems

1. .... Assignment and Derived Classes

This problem examines the difference between pointer assignment and 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. We can change a pointer to an object on the heap to point to a different object on the heap using pointer assignment. We can also assign an object on the stack to another object on the stack using object assignment, which causes the data of the object on the right side of the assignment to be copied into the data of the object on the left.

Consider the following code:

```
class Base {
  private:
    int x;
  public:
    Base(int i);
    virtual void f();
} ;
class Derived : public Base {
  private:
    int y;
  public:
    Derived(int i, int j);
    virtual void f();
} ;

Base::Base(int i){x=i;}
void Base::f(){x++;}

Derived::Derived(int i, int j) : Base(i) {y = j; }
void Derived::f(){Base::f();y++;}

void pointerAssign() {
  Base *b1 = new Base(0);
  Derived *d1 = new Derived(1,2);
  b1 = d1;
  b1->f();
}
void objectAssign() {
  Base b2(0);
  Derived d2(1,2);
  b2 = d2;
  b2.f();
}
int main() {
  pointerAssign();
```

```
  objectAssign();
}
```

(a) Draw the state of memory (stack, heap, and vtables) after pointers `b1` and `d1` have been assigned to objects in the call to `pointerAssign`.

(b) Redraw the relevant parts of memory from part (a) to reflect the changes that result after the assignment `b1=d1` occurs.

(c) Draw the state of memory (stack, heap, and vtables) after objects `b2` and `d2` have been allocated in the call to `objectAssign`.

(d) Redraw the state of memory from part (c) to reflect the changes that result after the assignment `b2=d2` occurs. Why isn't all of `d2`'s member data copied into `b2`?

(e) Why isn't `b2`'s vtable pointer changed by the assignment in `objectAssign`?

(f) Explain the difference between the code executed by the call `b1->g()` in `pointerAssign` and the code executed by the call `b2.g()` in `objectAssign`. Why do yoou think C++ works this way?

2. .... Phantom Members

A C++ class may have virtual members that may be redefined in derived classes. However, there is no way to "undefine" a virtual (or non-virtual) member. Suppose we extend C++ by adding another kind of member, called a *phantom* member, that is treated as virtual, but only defined in derived classes if an explicit definition is given. In other words, a "phantom" function is not inherited unless its name is listed in the derived class. For example, if we have two classes

```
class A {
...
public:
   phantom void f(){...}
   ...
};
class B : public A {
...
public:
   ... /* no definition of f */
};
```

then `f` would appear in the `vtbl` for `A` objects and, if `x` is an `A` object, `x.f()` would be allowed. However, if `f` is not declared in `B`, then `f` might not need to appear in the `vtbl` for `B` objects and, if `x` is a `B` object, `x.f()` would not be allowed. Is this consistent with the design of C++, or is there some general property of the language that would be destroyed? If so, explain what this property is, and why it would be destroyed.

3. .... Subtyping and Visibility

In C++, a virtual function may be given a different access level in a derived class. This produces some confusing situations. For example, this is legal C++, at least for some compilers:

```
class Base {
   public:
      virtual int f();
};
class Derived: public Base {
   private:
      virtual int f();
};
```

This question asks you to explain why this program conflicts with some reasonable principles, yet somehow does not completely break the C++ type system.

(a) In C++, a derived class `D` with public base class `B` is treated as a subtype of `B`. Explain why this is generally reasonable, given the definition of subtyping from class.

(b) Why would it be reasonable for someone to argue that it is *incorrect* to allow a public member inherited from a public base class to be redefined as private?

(c) A typical use of subtyping is to apply a function that expects an `B` argument to a `D`, when `D <: B`. For example, here is a simple "toy" program that applies a function defined for base class objects to a derived class object. Explain why this program *compiles* and *executes* for the `Base` and `Derived` classes above (assuming we have given an implementation for `f`).

```
int g(Base &x) {
  return(x.f()+1);
}

int main() {
  Base b;
  cout << "g(b) = " << g(b) << endl;
  Derived d;
  cout << "g(d) = " << g(d) << endl;
}
```

(d) Do you think there is a mistake here in the design of C++? Briefly explain why or why not.

4. .... "Like Current" in Eiffel

Eiffel is a statically-typed object-oriented programming language designed by Bertrand Meyer and his collaborators. The language designers did not intend the language to have any type loopholes. However, there are some problems surrounding an Eiffel type expression called `like current`. When the words `like current` appear as a type in a method of some class, they mean, "the class that contains this method" To give an example, the following classes were considered statically type correct in the language Eiffel.

```
Class Point
   x : int
   method equals (pt : like current) : bool
      return self.x == pt.x

class ColPoint inherits Point
   color : string
   method equals (cpt : like current) : bool
      return self.x == cpt.x and self.color == cpt.color
```

In `Point`, the expression `like current` means the type `Point`, while in `ColPoint`, `like current` means the type `ColPoint`. However, the type checker accepts the redefinition of method `equals` because the declared parameter type is `like current` in both cases. In other words, the declaration of `equals` in `Point` says that the argument of `p.equals` should be of the same type as `p`, and the declaration of `equals` in `ColPoint` says the same thing. Therefore, the types of `equals` are considered to match.

(a) Using the basic rules for subtyping objects and functions, explain why `ColPoint` should not be considered a subtype of `Point` "in principle."

(b) Give a short fragment of code that shows how a type error can occur if we consider `ColPoint` to be a subtype of `Point`.

(c) When this error was pointed out (by W. Cook after the language had been in use for several years), the Eiffel designers decided not to remove `like current`, since this would "break" lots of existing code. Instead, they decided to modify the type checker to perform whole-program analysis. More specifically, the modified Eiffel type checker examined the whole Eiffel program to see if there was any statement that was likely to cause a type error. Suppose you were trying to design a type checker that allows safe uses of `like current`. What kind of statements or expressions would your type checker look for? How would you distinguish a type error from a safe use of `like current`? Describe in a sentence or two how your type checker would prevent erroneous calls to `equals`.

5. .... 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 chapter 12 of the textbook.

(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?

6. .... 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?