
Reading

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

Problems

1. Delegation-Based OO Languages

In this problem, we explore a delegation-based object-oriented language *SELF* in which objects can be defined directly from other objects. Classes are not needed and not supported. *SELF* has run-time type-checking and garbage collection.

The *SELF* language description says:

In Smalltalk, ... everything is an object and every object contains a pointer to its class, an object that describes its format and holds its behavior. In *SELF* too, everything is an object. But, instead of a class pointer, a *SELF* object contains named slots with may store either state or behavior. If an object receives a message and it has no matching slot, the search continues via a *parent* pointer. This is how *SELF* implements inheritance. Inheritance in *SELF* allows objects to share behavior, which in turn allows the programmer to alter the behavior of many objects with a single change. For instance, a [Cartesian] point object would have slots for its non-shared characteristics: *x* and *y*. Its parent would be an object that held the behavior shared among all points: *+*, etc.

In *SELF*, there is no direct way to access a variable: instead, objects send messages to access data residing in name slots. So to access its “*x*” value, a point sends itself the “*x*” message. The message finds the “*x*” slot, and evaluates the object found therein. ... In order to change contents of the “*x*” slot to, say, 17, instead of performing an assignment like “*x*←17,” the point must send itself the “*x:*” message with 17 as the argument. The point object must contain a slot named “*x:*” containing the assignment [function].

- (a) Using this description, draw the run-time data structure of the *SELF* version of a (3,4) Point object, as described in the last two sentences of the first paragraph, and its parent object. Assume assignments to the *x* and *y* characteristics are permitted.
- (b) *SELF*'s lack of classes and instance variables make inheritance more powerful. For example, to create two points sharing the same “*x*” coordinate, the “*x*” and “*x:*” slots can be put in a separate object that is a parent of each of the two points. Draw a picture of the run-time data structures for two points sharing the same “*x*” coordinate.
- (c) To create a new Point object, the `clone` message is sent to an existing point. The language description continues:

Creating new objects ... is accomplished by a simple operation, copying. ... [In Smalltalk,] creating new objects from classes is accomplished by instantiation, which includes the interpretation of format information in a class. Instantiation is similar to building a house from a plan.

Cloning an object does not clone its parent, however.

If a *point* object contains fields *x* and *y*, and methods *x:*, *y:*, `move`, then cloning the object will create another object with two fields and three methods. Each point will have a parent

pointer, two fields, and three methods – six entries in all. The SELF point will be twice the size of the corresponding Smalltalk point.

Explain how you would structure your SELF program so that each point can be cloned without cloning its methods?

- (d) SELF also allows a *change parent* operation on an object. The parent pointer of an object can be set to point to any other object. (An exception is that the first object must not be an ancestor of the second – we don't want a cycle.)

The change parent operation is useful for objects that change behavior in different states. For example, a window can be in the *visible* or *iconified* (minimized) state. When iconified, mouse clicks and window display work differently than when the window is visible. A window's parent pointer can be set to `VisibleWindow` initially, then changed to `IconifiedWindow` when the "minimized" button is pressed. Another example is a file object that can be in the *open* or *closed* state. The `open` method changes the parent pointer from `ClosedFile` to `OpenFile`.

The change parent operation adds a lot of flexibility to SELF. Can you think of disadvantages of this feature?

2. C++ Casts and Object Representation

Occasionally programmers find it useful to convert variables from one type to another. In C++, this is accomplished using casts: `reinterpret_cast<T>`, `static_cast<T>`, and `dynamic_cast<T>`. For simplicity, we limit our discussion to casts of object pointers.

- The most permissive cast is the `reinterpret_cast<T>`, which simply changes the compile-time type of a pointer to `T` regardless of the runtime type of the object.
- The `static_cast`, the next most permissive, converts from `X*` to `Y*` if `X` is either subclass or a superclass of `Y`, ignoring the runtime type of the object being cast. This conversion might change the value of the pointer in cases with multiple inheritance.
- The `dynamic_cast` performs the same type conversions as `static_cast`, but checks the runtime type of the converted object to ensure that the compile-time type is a supertype of the runtime type. It returns null if the check fails.

This problem examines the behavior of these casts as complicated by the runtime representation of objects in C++.

- (c) When a compiler converts a source program into object code, sometimes the compiler needs to generate code to implement a cast. Consider a machine with the following three instructions.

```
movl  src, dst          ; move source src into destination dst
addl  src, dst          ; increment dst by src
subl  src, dst          ; decrement dst by src
```

For each instruction, `dst` names a register and `src` names either a register or a constant prefixed by a `$`. For example, `addl $12, %eax` adds 12 to the contents of register `%eax`. Assuming that `a` is stored in register `%eax`, `b` is stored in register `%ebx`, and `c` is stored in register `%ecx`, please match each of the following casts to its most likely assembly implementation: (Note: some assembly sequences might not be used.) Assume that object pointer point to the lowest address of the object.

```
class A {...};
class B {...};
class C: public A, public B {...};
```

```
A *a; B *b; C *c;
```

```
c = new C();
a = static_cast<A*>(c);          // Implemented by asm _____
```

```

b = static_cast<B*>(c); // Implemented by asm _____
c = reinterpret_cast<C*>(b); // Implemented by asm _____
c = static_cast<C*>(b); // Implemented by asm _____
a = reinterpret_cast<A*>(c); // Implemented by asm _____
b = reinterpret_cast<B*>(c); // Implemented by asm _____
c = static_cast<C*>(a); // Implemented by asm _____

```

```

(i)   movel %eax, %ecx
(ii)  movel %ebx, %ecx
(iii) movel %ecx, %eax
(iv)  movel %ecx, %ebx
(v)   movel %ecx, %eax
      addl  $12, %eax
(vi)  movel %ecx, %ebx
      addl  $12, %ebx
(vii) movel %ecx, %eax
      subl  $12, %eax
(viii) movel %ecx, %ebx
      subl  $12, %ebx
(ix)  movel %eax, %ecx
      addl  $12, %ecx
(x)   movel %ebx, %ecx
      addl  $12, %ecx
(xi)  movel %eax, %ecx
      subl  $12, %ecx
(xii) movel %ebx, %ecx
      subl  $12, %ecx

```

- (b) Some legal instances of `static_cast` can result in memory errors. Fill in the blanks so that the following program can result in a memory error (and does not have a memory error if `static_cast` is replaced with `dynamic_cast`).

```

class Base {
public:
    int x;
    virtual void f( void ) { cout << "Base class" << endl; }
};
class Derived: public Base {
public:
    int y;
    virtual void f( void ) { cout << "Derived class" << endl; }
};

int main( void ) {
    _____ p = static_cast<_____>( _____ );
    if( p ) {
        p->_____
    }

    return 0;
}

```

- (c) The `dynamic_cast` can be applied only to objects whose compiletime class contains at least one virtual function. Your friend complains that this is overly restrictive and suggests that the cast should be applicable without this restriction. Please impress your friend with your knowledge of the runtime representation of objects in C++ by explain why this restriction is necessary. (Your friend is also impressed by brevity, so please limit your explanation to three sentences.)
- (d) Java has only one kind of cast. Of the three casts, which is most similar to a cast in Java? In one sentence, explain why Java chose to implement that cast.

3. Function Subtyping

Assume that `Square <: Rectangle` and `Rectangle <: Shape`. Which of the following subtype relationships hold in principle?

- i) `(Rectangle → Rectangle) <: (Rectangle → Rectangle)`
- ii) `(Rectangle → Square) <: (Rectangle → Rectangle)`
- iii) `(Rectangle → Shape) <: (Rectangle → Rectangle)`
- iv) `(Shape → Rectangle) <: (Rectangle → Rectangle)`
- v) `(Square → Rectangle) <: (Rectangle → Rectangle)`
- vi) `(Shape → Square) <: (Rectangle → Rectangle)`
- vii) `(Square → Square) <: (Rectangle → Rectangle)`
- viii) `(Shape → Shape) <: (Rectangle → Rectangle)`
- ix) `(Square → Rectangle) → Shape <: (Rectangle → Square) → Shape`
- x) `(Square → Rectangle) → Shape <: (Square → Rectangle) → Rectangle`
- xi) `(Square → Square) → Square <: (Rectangle → Rectangle) → Rectangle`
- xii) `(Square → Square) → Square <: (Rectangle → Square) → Rectangle`

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) Suppose `Point` has a `move` method that requires integer arguments, and returns a result of type `like current`. Assume that this method is inherited in subclass `ColPoint`. Write the type of `move` in `ColPoint` two ways, once using `like current` and once replacing `like current` with the type that this refers to (`Point` or `ColPoint`).
- (b) If `ColPoint` is a subtype of `Point`, is the type of `move` in `ColPoint` a subtype of the type of `move` in `Point`? Explain.
- (c) Write the type of `equals` in `ColPoint` two ways, once using `like current` and once replacing `like current` with the type that this refers to (`Point` or `ColPoint`).
- (d) If `ColPoint` is a subtype of `Point`, is the type of `equals` in `ColPoint` a subtype of the type of `equals` in `Point`? Explain.
- (e) The designers of Eiffel wanted the subclass `ColPoint` to be a subtype of `Point`. Do you think they made some kind of mistake here? Explain.

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 this chapter.

- 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()`?
- 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.
- 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?
- How could the call `pc->g()` be implemented?
- Multiple inheritance also adds some complication to the C++ casting system. The standard C casts suddenly do not work as expected for inherited classes. Examine the following C++ code:

```
#include <stdio.h>

class Object{
    public:
        virtual int Hash() {return (int)this;}
};
class A:public Object{
    public:
        int x;
        A(){x=1;}
};
```

```

class B:public Object{
    public:
        int y;
        B(){y=2;}
};
class C:public A,public B{
    public:
        int z;
        C() {z=3;}
};

int main () {
    C * c= new C;
    A * aP=c;
    B * bP=c;
    printf ("aP.x %d bP.y: %d\n",aP->x ,bP->y);
    {
        C * cP=(C*)(void*)aP;
        printf ("cP.x: %d cP.y: %d\n",cP->x,cP->y);
    }
    C * cP=(C*)(void*)bP;
    printf ("cP.x: %d cP.y: %d\n",cP->x,cP->y);

    C * cPgood= (C*)bP;
    printf ("dyncP.x: %d dyncP.y: %d\n",cPgood->x, cPgood->y);
    return 0;
}

```

Write what gets printed out in this example, and explain in 1 sentence the anomaly that occurs.

Draw a picture of how C looks (each field) and where aP, bP, cP, and cPgood point within the class.

Explain in 1 sentence what steps have to be accomplished to perform the correct cast in the last print statement.

- (f) The Java object system forces all classes to inherit from an Object class. This example illustrates one reason inheriting from Object can cause problems in C++, a language with multiple inheritance. What trouble occurs if you write:

```

void blah (Object * reallyC) {
    C* cP = (C*)(reallyC);
    printf ("cP.x: %d cP.y: %d\n",cP->x,cP->y);
}

```

In one sentence, why does this occur? Does this problem occur at compile time or run time? At a high level, explain in 1 sentence how one could add a new casting operation to make the “blah” function work. This cast is known as a “dynamic_cast” and operates as follows:
C * cP = dynamic_cast<C*>(reallyC);

Explain in 2 sentences at a low level what a dynamic_cast must do for the function to operate properly.

- (g) Why does the C++ compiler complain if you call c->Hash() in main()? Does aP->Hash() return the same value as bP->Hash()? Why or why not? Does c->A::Hash() return the same value as c->B::Hash()? Why or why not?
- (h) What happens to the values returned by c->Hash(), aP->Hash() and bP->Hash() when you change A and B’s inheritance of Object from “public” to “virtual public” as follows:

```
class A:virtual public Object{...};
class B:virtual public Object{...};
```

In one sentence, how is the problem resolved in this particular case?

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 obvious 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.

- (c) We may generalize this notion of dispatch based 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?