

---

## Reading

---

1. Read Chapter 9, Data Abstraction and Modularity, skipping section 9.2.5 on datatype induction and the material on ML modules and CLU abstraction mechanisms (unless you are interested).
2. Read Chapter 10, Concepts in Object-Oriented Languages.
3. Read Chapter 11 on Simula and Smalltalk.

---

## Problems

---

### 1. .... Function objects in STL

In STL terminology, a *function object* is any object that can be called as if it is a function. Any object of a class that defines `operator()` is a function object. In addition, an ordinary function may be used as a function object, since `f()` is meaningful if `f` is a function, and similarly if `f` is a function pointer.

Here is a C++ template for combining an argument type, return type, and `operator()` together into a function object. Every object from every subclass of `FuncObj<A,B>`, for any types `A` and `B` is a function object, but not all function objects come from such classes.

```
template <typename Arg, typename Ret>
class FuncObj {
public:
    typedef Arg argType;
    typedef Ret retType;
    virtual Ret operator()(Arg) = 0;
};
```

Here are two example classes of function objects. In the first case, the constructor stores a value in a protected data field so that different function objects from this class will divide by different integers. In a sense to be explored later in this problem, instances of `DivideBy` are similar to closures since they may contain hidden data.

```
class DivideBy : public FuncObj<int, double> {
protected:
    int divisor;
public:
    DivideBy(int d) {
        this->divisor = d;
    }
    double operator()(int x) {
        return x/((double)divisor);
    }
};
```

```
class Truncate : public FuncObj<double, int> {
public:
```

```

int operator()(double x) {
    return (int) x;
}
};

```

Since DivideBy uses the FuncObj template as a public base class, DivideBy is a subtype of FuncObj<int, double>, and similarly for Truncate.

- (a) Fill in the blanks to complete the Compose template below. The Compose constructor takes two function objects f and g and creates a function object that computes their composition  $\lambda x. f(g(x))$ .

You will need to fill in type declarations on lines 9 and 13 and code on lines 10, 11, and 14. If you do not know the correct C++ syntax, you may use short English descriptions for partial credit. (We have double-checked the parentheses to make sure they are correct.)

To give you a better idea of how Compose is supposed to work, you may want to look at the sample code below that uses Compose to compose DivideBy and Truncate.

```

1: template < typename Ftype , typename Gtype >
2: class Compose :
3:     public FuncObj < typename Gtype::argType,
4:                   typename Ftype::retType > {
5: protected:
6:     Ftype *f;
7:     Gtype *g;
8: public:
9:     Compose( _____ f, _____ g) {
10:
11:         _____ = f ;
12:     }
13:
14:     _____ operator()( _____ x) {
15:         return ( _____ )(( _____ )( _____ ));
16:     };

```

```

void main() {
    DivideBy *d = new DivideBy(2);
    Truncate *t = new Truncate();
    Compose<DivideBy, Truncate> *c1
        = new Compose<DivideBy, Truncate>(d,t);
    Compose<Truncate, DivideBy> *c2
        = new Compose<Truncate, DivideBy>(t,d);

    cout << (*c1)(100.7) << endl; // Prints 50.0
    cout << (*c2)(11) << endl;    // Prints 5
}

```

- (b) Consider code of the following form, where  $A$ ,  $B$ ,  $C$ ,  $D$  are types that might not all be different (i.e., we could have  $A = B = C = D = \text{int}$  or  $A$ ,  $B$ ,  $C$ ,  $D$  might be four different types):

```
class F : public FuncObj<A, B> {
    ...
};
class G : public FuncObj<C, D> {
    ...
};
F *f = new F(...);
G *g = new G(...);
Compose<F, G> *h
    = new Compose<F, G>(f,g);

cout << (*h)(...) << endl; // call compose function
```

What will happen if the return type  $D$  of  $g$  is not the same as the argument type  $A$  of  $f$ ? If it is possible for an error to occur and possible for an error not to occur, say which conditions will cause an error and which will not. If it is possible for an error to occur at compile time or at run time, say when the error will occur and why.

- (c) Our `Compose` template is written assuming that `Ftype` and `Gtype` are classes that have `argType` and `retType` type definitions. If you wanted to define a `Compose` template that works for all function objects, what arguments would your template have and why? Your answer should be in the form of “the types of variables ..., the arguments of functions ..., and the return values of ...”. Assume that the code in lines 10, 11, and 14 stays the same. All we want to change are the template parameters on line one and possibly the parts of the body of the template that refer to template parameters.

## 2. .... Expression Objects

We can represent expressions given by the grammar

$$e ::= \text{num} \mid e + e$$

using objects from a class called `expression`. We begin with an “abstract class” called `expression`. While this class has no instances, it lists the operations common to all kinds of expressions. These are a predicate telling whether there are subexpressions, the left and right subexpressions (if the expression is not atomic), and a method computing the value of the expression.

```
class expression() =
    private fields:
        (* none appear in the _interface_ *)
    public methods:
        atomic?() (* returns true if no subexpressions *)
        lsub()    (* returns ``left`` subexpression if not atomic *)
        rsub()    (* returns ``right`` subexpression if not atomic *)
        value()  (* compute value of expression *)
end
```

Since the grammar gives two cases, we have two subclasses of `expression`, one for numbers and one for sums.

```
class number(n) = extend expression() with
    private fields:
        val num = n
```

```

    public methods:
        atomic?() = true
        lsub    () = none  (* not allowed to call this, *)
        rsub    () = none  (* because atomic?() returns true *)
        value   () = num
end

class sum(e1,e2) = extend expression() with
    private fields:
        val left = e1
        val right = e2
    public methods:
        atomic?() = false
        lsub    () = left
        rsub    () = right
        value   () = ( left.value() ) + ( right.value() )
end

```

(a) *Product Class*

Extend this class hierarchy by writing a `prod` class to represent product expressions of the form

$$e ::= \dots \mid e * e$$

(b) *Method Calls*

Suppose we construct a compound expression by

```

val a = number(3);
val b = number(5);
val c = number(7);
val d = sum(a,b);
val e = prod(d,c);

```

and send the message `value` to `e`. Explain the sequence of calls that are used to compute the value of this expression: `e.value()`. What value is returned?

(c) *Unary Expressions*

Extend this class hierarchy by writing a `square` class to represent squaring expressions of the form

$$e ::= \dots \mid e^2$$

What changes will be required in the expression interface? What changes will be required in subclasses of `expression`? What changes will be required in functions that use expressions? What changes will be required in functions that do not use expressions? (Try to make as few changes as possible to the program.)

*Hint:* consider the following function that counts the number of leaves in the parse tree of the expression.

```

fun count (e)
    if e.atomic?() then return 1
    else return (count(e.lsub()) + count (e.rsub()))
end

```

(d) *Ternary Expressions*

Extend this class hierarchy by writing a `cond` class to represent conditionals\* of the form

$$e ::= \dots \mid e ? e : e$$

---

\*In C, conditional expressions `a?b:c` evaluate `a`, and then return the value of `b` if `a` is non zero, or return the value of `c` if `a` is zero.

What changes will be required if we wish to add this ternary operator? (As in part (c), try to make as few changes as possible to the program.)

(e) *N-Ary Expressions*

Explain what kind of interface to expressions we would need if we would like to support atomic, unary, binary, ternary and  $n$ -ary operators without making further changes to the interface. In this part of the problem, we are not concerned with minimizing the changes to the program; instead, we are interested in minimizing the changes that may be needed in the future.

### 3. .... Objects vs. Type Case

With object oriented programming, classes and objects can be used to avoid “type case” statements. Here is a program using a form of case statement that inspects a user-defined type tag to distinguish between different classes of shape objects. This program would not statically type-check in most typed languages since the correspondence between the tag field of an object and the class of the object is not statically guaranteed and visible to the type checker. However, in an untyped language like Smalltalk, a program like this could behave in a computationally reasonable way.

```
enum shape_tag { s_point, s_circle, s_rectangle };

class point {
    shape_tag tag;
    int x;
    int y;

    point (int xval, int yval)
        { x = xval; y = yval; tag = s_point; }
    int x_coord () { return x; }
    int y_coord () { return y; }
    void move (int dx, int dy) { x += dx; y += dy; }
};

class circle {
    shape_tag tag;
    point c;
    int r;

    circle (point center, int radius)
        { c = center; r = radius; tag = s_circle }
    point center () { return c; }
    int radius () { return radius; }
    void move (int dx, int dy) { c.move (dx, dy); }
    void stretch (int dr) { r += dr; }
};

class rectangle {
    shape_tag tag;
    point tl;
    point br;

    rectangle (point topleft, point botright)
        { tl = topleft; br = botright; tag = s_rectangle; }
    point top_left () { return tl; }
    point bot_right () { return br; }
```

```

void move (int dx, int dy) { tl.move (dx, dy); br.move (dx, dy); }
void stretch (int dx, int dy) { br.move (dx, dy); }
};

/* Rotate shape 90 degrees. */
void rotate (void *shape) {
  switch ((shape_tag *) shape) {
    case s_point:
    case s_circle:
      break;
    case s_rectangle:
      {
        rectangle *rect = (rectangle *) shape;
        int d = ((rect->bot_right ().x_coord ()
                  - rect->top_left ().x_coord ()) -
                 (rect->top_left ().y_coord ()
                  - rect->bot_right ().y_coord ()));
        rect->move (d, d);
        rect->stretch (-2.0 * d, -2.0 * d);
      }
  }
}

```

- (a) Rewrite this so that instead of `rotate` being a function, each class has a `rotate` method, and the classes do not have a `tag`.
- (b) Discuss, from the point of view of someone maintaining and modifying code, the differences between adding a triangle class to the first version (as written above) and adding a triangle class to the second (produced in part (a) of this question).
- (c) Discuss the differences between changing the definition of `rotate` (say, from 90 degrees to the left to 90 degrees to the right) in the first and second versions. Assume you have added a triangle class so that there is more than one class with a nontrivial `rotate` method.

#### 4. .... Simula Inheritance and Access Links

In Simula, a class is a procedure that returns a pointer to its activation record. Simula prefixed classes are a precursor to C++ derived classes, providing a form of inheritance. This question asks about how inheritance might work in an early version Simula, assuming that the standard static scoping mechanism associated with activation records is used to link the derived class part of an object with the base class part of the object.

Sample `Point` and `ColorPt` classes are given in the text (Section 11.2). For the purpose of this problem, assume that if `cp` is a `ColorPt` object, consisting of a `Point` activation record followed by a `ColorPt` activation record, the access link of the parent class (`Point`) activation record points to the activation record of the scope in which the class declaration occurs, and the access link of the child class (`ColorPt`) activation record points to activation record of the parent class.

- (a) Fill in the missing information in the following activation records, created by executing the following code:

```

ref(Point) r;
ref(ColorPt) cp;
r :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);
cp.distance(r);

```

Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled “access link.” The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

<i>Activation Records</i>	<i>Closures</i>	<i>Compiled Code</i>
(0)		
(1) Point(...)		
(2) Point part of cp		
(3) ColorPt(...)		
(4) cp.distance(r)		

(b) The body of `distance` contains the expression

$$\text{sqrt}((x - q.x) ** 2 + (y - q.y) ** 2)$$

which compares the coordinates of the point containing this `distance` procedure to the coordinate of the point `q` passed as an argument. Explain how the value of `x` is found when `cp.distance(r)` is executed. Mention specific pointers in your diagram. What value of `x` is used?

- (c) This illustration shows that a reference `cp` to a colored point object points to the `ColorPt` part of the object. Assuming this implementation, explain how the expression `cp.x` can be evaluated. Explain the steps used to find the right `x` value on the stack, starting by following the pointer `cp` to activation record (3).
- (d) Explain why the call `cp.distance(r)` only needs access to the `Point` part of `cp` and not the `ColorPt` part of `cp`.
- (e) If you were implementing Simula, would you place the activation records representing objects `r` and `cp` on the stack, as shown here? Explain briefly why you might consider allocating memory for them elsewhere.

## 5. .... Smalltalk Run-time Structures

Here is a Smalltalk `Point` class whose instances represent points in the two-dimensional Cartesian plane. In addition to accessing instance variables, an instance method allows point objects to be added together.

class name	Point
superclass	Object
class variables	<i>comment: none</i>
instance variables	x y
class messages and methods	<i>comment: instance creation</i> <b>newX: xValue Y: yValue</b>    ↑ self new x: xValue y: yValue
instance messages and methods	<i>comment: accessing instance vars</i> <b>x: xCoordinate y: yCoordinate</b>    x ← xCoordinate y ← yCoordinate <b>x</b>    ↑ x <b>y</b>    ↑ y <i>comment: arithmetic</i> <b>+ aPoint</b>    ↑ Point newX: (x + aPoint x) Y: (y + aPoint y)

- (c) Complete the top half of the drawing of the Smalltalk run-time structure shown in Figure 1 for a point object with coordinates (3,4) and its class. Label each of the parts of the top half of the figure, adding to the drawing as needed.
- (b) A Smalltalk programmer has access to a library containing the `Point` class, but she cannot modify the `Point` class code. In her program, she wants to be able to create points using either cartesian or polar coordinates, and she wants to calculate both the polar coordinates (radius and angle) and the Cartesian coordinates of points. Given a point  $(x, y)$  in cartesian coordinates, the radius is  $((x * x) + (y * y))$  `squareRoot`, and the angle is  $(x/y)$  `arctan`. Given a point  $(r, \theta)$  in polar coordinates, the  $x$  coordinate is  $r * (\theta \text{ cos})$  and the  $y$  coordinate is  $r * (\theta \text{ sin})$
- Write out a subclass, `PolarPoint`, of `Point` and explain how this solves the programming problem.
  - Which parts of `Point` could you reuse and which would you have to define differently for `PolarPoint`?
- (c) Complete the drawing of the Smalltalk run-time structure by adding a `PolarPoint` object and its class to the bottom half of the figure you already filled in with `Point` structures. Label each of the parts and add to the drawing as needed.

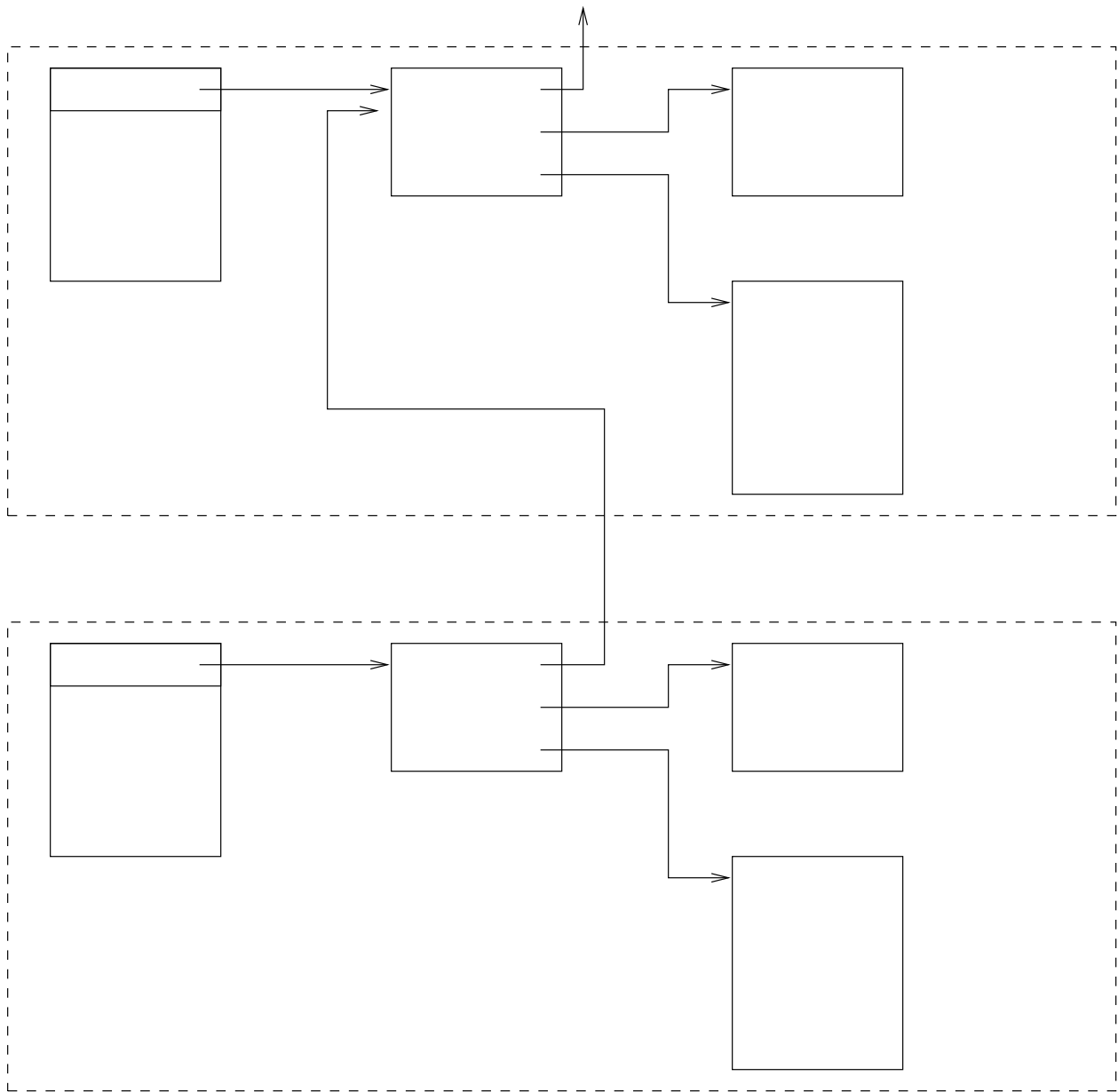


Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint