Homework 5

Due 12 November

Handout 13
CS242: Autumn 2003
5 November

Reading

1. Read Chapter 9, Data Abstraction and Modularity, skipping section 9.2.5 on datatype induction.
2. Read Chapter 10, Concepts in Object-Oriented Languages.

Problems

1. .... Function Templates and Overloading

As we discussed in connection with polymorphism, function templates are a way to add type variables to the C++ type system. In the example below, the type `T` allows the function `f` to take different types of arguments without changing the function body. The declarations on lines (a)–(c) are referred to as *function base templates*:

```
template<class T> void f( T );              // (a)
template<class T> void f( int, T, double ); // (b)
template<class T> void f( T* );             // (c)
void f( double );                           // (d)

int main() {
long l;
int i;
double d;

f( l );         // calls (a) with T = long
f( i, 42, d );  // calls (b) with T = int
f( &i );        // calls (c) with T = int
f( i );         // calls (a) with T = int
f( d );         // calls (d)
}
```

The complication with function templates and overloading arises when we have multiple versions of a function that can match a single call. In the absence of templates, the C++ compiler prevents this sort of clash. With templates, though, we can see that the declarations (a) and (d) clash, but the C++ compiler does not complain about this code. For example, the call `f( d )` on the last line of `main` could resolve to either declaration (a) or declaration (d).

Here are some rules for resolving overloaded function templates:

(a) Nontemplate functions are preferred. If a nontemplate function matches the parameter types as well as any function template, the nontemplate function is used.

(b) If there are no nontemplate functions that are at least as good, then a function base template will be used. Which function base template gets selected depends on which matches best and is the "most specialized," according to a set of fairly arcane rules:

   i. If there is one "most specialized" function base template, that one gets used.

   ii. If there is a tie for the "most specialized" function base template, the call is ambiguous because the compiler cannot decide which is a better match. The programmer will have to do something to qualify the call and say which one is wanted.

iii. If there is no function base template that can be made to match, the call is bad and the programmer will have to fix the code.

We can provide an intuitive definition of "specialization" by saying that one template is more specialized than another if it can match fewer types. For example, the function base template on line (c) is more specialized than the function base template on line (a).

(a) (5 points) For each of the calls in the example above, write down which of the rules above is used to resolve clashes in overload resolution. If there is no clash (i.e. only one choice among (a)-(d) is possible), write "no clash." Otherwise write the specific rule that is used (e.g., (b)-iii, would refer to the very last rule).

(b) Given that the call to `f(d)` could be handled by the declaration on line (a), why might a programmer choose to overload `f` with the declaration on line (d)? More specifically, write a body for the declaration on line (a) that contains no more than one line and will cause the C++ type checker to report an error if you eliminated the declaration on line (d). In writing your definition, remember that C and C++ will often implicitly cast the parameters to an operator.

```
template<class T> void f(T x) { ____________________________; }
```

2. .... 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 is `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 $\lambda x.\, \mathtt{f}(g(x))$.

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:    ____________ = f ;

11:    ____________ = g ;
12:  }

13:  ____________ operator()( ____________ x) {

14:    return ( ______ )(( ______ )( ______ ));
15:  }
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 = 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.

The rest of this problem asks about the similarities and differences between C++ function objects and function closures in languages like Lisp, ML, and Scheme. In studying scope and activation records, we looked at a function makeCounter that returns a counter, initialized to some integer value. Here is makeCounter function in Scheme:

```
(define makeCounter
  (lambda (val)
    (let (( counter (lambda (inc) (set! val (+ val inc)) val)) )
      counter)))
```

Here is part of an interactive session using makeCounter

```
==> (define c (makeCounter 3))
#<unspecified>
==> (c 2)
5
==> (c 2)
7
```

The first input defines a counter c, initialized to 3. The second input line adds 2, producing value 5, and the third input line adds 2 again, producing value 7.

A general idea for translating Lisp functions into C++ function objects is to translate each function f into a class A so that objects from class A are function objects that behave like f. If f is defined within some nested scope, then the constructor for A will put the global variables of f into the function object, so that an instance of A behaves like a closure for f.

Since makeCounter does not have any free variables, we can translate makeCounter into a class MAKECOUNTER that has a constructor with no parameters.

```
class MAKECOUNTER {
    public:
        class COUNTER {
            protected:
                int val;
            public:
                COUNTER(int init){val = init;}
                int operator()(int inc) {
                    val = val + inc;
                    return val;
                 }
        };
        MAKECOUNTER(){}
        COUNTER* operator()(int val) {
             return new COUNTER(val);
         }
};
```

We can create a MAKECOUNTER function object and use it to create a COUNTER function object as follows.

```
MAKECOUNTER *m = new MAKECOUNTER();
MAKECOUNTER::COUNTER * c = (*m)(3);

cout << (*c)(2) << endl; // Prints 5
```

(d) Thinking generally about Lisp closures and C++ function objects, describe one way in which C++ function objects might serve some programming objectives better than Lisp closures.

(e) Thinking generally about Lisp closures and C++ function objects, describe one way in which Lisp closures might serve some programming objectives better than C++ function objects.

(f) Do you think this approach sketched in this problem will allow you to translate an arbitrary nesting of Lisp or ML functions into C++ function objects? You may want to consider the following variation of MAKECOUNTER in which the counter value val is placed in the outer class instead of the inner class.

```
class MAKECOUNTER {
protected:
  int val;

public:
  class COUNTER {
  private:
    MAKECOUNTER *mc;

  public:
    COUNTER(MAKECOUNTER* mc, int init) {
      this->mc = mc;
      mc->val = init;
    }

    int operator()(int inc) {
      mc->val = mc->val + inc;
      return mc->val;
    }
  };
  friend class COUNTER;

  MAKECOUNTER(){}
  COUNTER* operator()(int val) {
    return new COUNTER(this,val);
  }
};
```

3. .... Expression Objects

We can represent expressions given by the grammar

$$e \quad ::= \quad num \mid e + e$$

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 commons 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

$$e \quad ::= \quad \ldots \mid e * e$$

(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

$$e \quad ::= \quad \ldots \mid e^2$$

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.)

(d) *Ternary Expressions*

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

$$e \quad ::= \quad \ldots \mid e?e : e$$

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.

4. .... 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 += dy; y += dy; }
};

class circle {
  shape_tag tag;
```

*Keep in mind that not all functions simply want to evaluate entire expressions. They may call the other methods as well.

†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.

```
  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.

5. .... Visitor Design Pattern

The extension and maintenance of an object hierarchy can be greatly simplified (or greatly complicated) by design decisions made early in the life of the hierarchy. This question explores various design possibilities for an object hierarchy representing arithmetic expressions.

The designers of the hierarchy have already decided to structure it as shown below, with a base class `Expression` and derived classes `IntegerExp`, `AddExp`, `MultExp` and so on. They are now contemplating how to implement various operations on Expressions, such as printing the expression in parenthesized form or evaluating the expression. They are asking you, a freshly-minted language expert, to help.

The obvious way of implementing such operations is by adding a method to each class for each operation. The Expression hierarchy would then look like:

```
class Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
}
class IntegerExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
}
class AddExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
}
```

Suppose there are n subclasses of `Expression` altogether, each similar to `IntegerExp` and `AddExp` shown here. How many classes would have to be added or changed to add each of the following things?

(a) A new class to represent product expressions.

(b) A new operation to graphically draw the expression parse tree.

Another way of implementing expression classes and operations uses a pattern called the Visitor Design Pattern. In this pattern, each operation is represented by a Visitor class. Each Visitor class has a `visitCLS` method for each expression class `CLS` in the hierarchy. The expression class `CLS` is set up to call the `visitCLS` method to perform the operation for that particular class. Each class in the expression hierarchy has an `accept` method which accepts a Visitor as an argument and "allows the Visitor to visit the class and perform its operation." The expression class does not need to know what operation the visitor is performing.

If you write a Visitor class `ParenPrintVisitor` to print an expression tree, it would be used as follows:

```
    Expression *expTree = ...some code that builds the expression tree...;
    Visitor *printer = new ParenPrintVisitor();
    expTree->accept(printer);
```

The first line defines an expression, the second defines an instance of your `ParenPrintVisitor` class, and the third passes your visitor object to the `accept` method of the expression object.

The expression class hierarchy using the Visitor Design Pattern has this form, with an `accept` method in each class and possibly other methods.

```
class Expression
{
    virtual void accept(Visitor *vis) = 0; //Abstract class
    //...
}
class IntegerExp : public Expression
{
    virtual void accept(Visitor *vis) {vis->visitIntExp(this);};
    //...
}
class AddExp : public Expression
{
    virtual void accept(Visitor *vis)
    { lhs->accept(vis); vis->visitAddExp(this); rhs->accept(vis); }
    //...
}
```

The associated `Visitor` abstract class, naming the methods that must be included in each visitor, and some example subclasses, have this form:

```
class Visitor
{
    virtual void visitIntExp(IntegerExp *exp) = 0;
    virtual void visitAddExp(AddExp *exp) = 0;   // Abstract class
}

class ParenPrintVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp print code};
    virtual void visitAddExp(AddExp *exp) {// AddExp print code};
}

class EvaluateVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp eval code};
    virtual void visitAddExp(IntegerExp *exp) {// AddExp eval code};
}
```

Suppose there are n subclasses of `Expression`, and m subclasses of `Visitor`. How many classes would have to be added or changed to add each of the following things using the Visitor Design Pattern?

(c) A new class to represent product expressions.

(d) A new operation to graphically draw the expression parse tree.

The designers want your advice.

(e) Under what circumstances would you recommend using the standard design?

(f) Under what circumstances would you recommend using the Visitor Design Pattern?