

Constructors, destructors, ...

```
class Exp {
private:
    int kind; // 1: number; 2: product; 3: sum
    int val; // if kind is 1
    Exp* left; Exp* right; // if kind is 2/3
public:
    Exp(int v);
    Exp(Exp* e1, Exp* e2, int k);
    ...}

Exp::Exp(int v)
    { kind = 1; val = v; left = 0; right = 0; }

Exp::Exp(Exp* e1, Exp* e2, int k)
    { if ((k < 2) || (k > 3)) throw("Trouble!");
      kind = k; left = e1; right = e2; }
```

Potential aliasing problems

```
Exp a(1); Exp b(2); Exp c(3); Exp d(4);
```

```
Exp p1(&a, &b, 2);
```

```
Exp s1(&a, &b, 3);
```

```
a = b; // Now p1, s1 will also change!
```

```
a = p1; // Even worse!
```

Moral: Assignment operation can cause serious trouble with objects that include pointers. Similar trouble with “copy construction”:

```
Exp p2 = p1; //copies members of p1 into p2
```

Solution: Define *copy constructor* and *assignment operator* appropriately.

Copy constructor

```
Exp::Exp(Exp& e1) { // par: by ref.  
    kind = e1.kind;  
    if (kind == 1) { val = e1.val; return; }  
    left = &Exp(*e1.left); // Deep copy  
    right = &Exp(*e1.right); // Deep copy  
    return; }
```

Assignment operator

```
Exp& Exp::operator=(Exp& e1) {  
    if (this == &e1) return *this; // self-copy  
    kind = e1.kind;  
    if (kind == 1) { val = e1.val; return *this; }  
    delete left; left = &Exp(*e1.left);  
    delete right; right = &Exp(*e1.right);  
    return *this; } //like the copy constructor
```

Computing values

```
int Exp::myVal( ) { //returns value
  if (kind == 1) return val;
  if (kind == 2)
    return((left->myVal()) * (right->myVal()));
  if (kind == 3)
    return((left->myVal()) + (right->myVal()));
  return 0; // ??
}
```

Inheritance, Polymorphism (Ch. 12)

The use of `kind` to distinguish between different kinds of `Exp` objects can cause problems.

Better solution: Inheritance + polymorphism.

Basic idea:

Introduce *base* `Exp` class;

The different kinds of expressions (numbers, sums, products) are *derived* classes of `Exp`.

In the base class, define the common stuff; in the derived classes, define the *specialized* stuff.

The *system* will keep track of what particular kind of `exp` we are dealing with.

Exp base class

```
class Exp {
private:
    int valCnt; // counts calls to myVal
protected:
    void incCount() { valCnt++; }
public:
    Exp() { valCount = 0; }
    virtual ~Exp() { } //destructor
    virtual int myVal() = 0; // ‘‘pure virtual’’
    // virtual int myVal() {return 0;} // virtual
    int valCount { return valCnt; }
};
```

Derived classes

```
class NumExp : public Exp {  
private:  
    int val; public:  
    NumExp(int i) : Exp(), val(i) { }  
    int myVal() { incCount(); return val; }
```

Derived classes (contd.)

```
class SumExp : public Exp {
private:
    Exp* left; Exp* right;
public:
    SumExp(Exp* e1, Exp* e2)
        : Exp() { left=e1; right=e2; }
    int myVal() { incCount();
        return ((left->myVal() + right->myVal())); }
};
```

```
class ProdExp : public Exp {
private:
    Exp* left; Exp* right;
public:
    ProdExp(Exp* e1, Exp* e2)
        : Exp() { left=e1; right=e2; }
    int myVal() { incCount();
        return ((left->myVal() * right->myVal())); }
};
```

Using the classes

```
Exp* expArr[10];  
expArr[0] = new NumExp(10);  
expArr[1] = new NumExp(20);  
expArr[2] = new NumExp(30);  
expArr[3] = new NumExp(40);  
  
expArr[4] = new SumExp(expArr[0], expArr[1]);  
expArr[5] = new ProdExp(expArr[2], expArr[3]);  
expArr[6] = new SumExp(expArr[4], expArr[5]);  
expArr[7] = new ProdExp(expArr[4], expArr[5]);
```

Using the classes (contd.)

```
for (i=0; i<8; i++) {  
    cout << expArr[i]->myVal(); }  
}
```

That will output the values of the four NumExp objects in elements 0 through 3, the two SumExp objects in elements 4, 6 and the two ProdExp objects in 5 and 7.

How does the system know which `myVal()` function to invoke?

More on Exp

```
int SumExp::myVal() {  
    incCount(); // keeps count of calls to myVal();  
    return (left->myVal() + right->myVal()); }  
}
```

The `left->myVal()` and `right->myVal()` calls are also dispatched to the appropriate `myVal()` functions based on the type of `Exp` `left` and `right` point to.

A New Kind of Exp

```
class InputExp : public Exp {
private:
    // Nothing!
public:
    InputExp()
        : Exp() { }
    int myVal() { incCount();
        int i; cin >> i; return i; } };
```

An InputExp is an “input expression”.

It reads in a new value whenever its myVal() is called.

Key point:

Now you can create a `SumExp` object one of whose components is an `InputExp` and the `myVal` function of `SumExp` works without change:

```
Exp* ip = new InputExp();  
Exp* np = new NumExp(5);  
  
Exp* sp = new SumExp(ip, np);  
cout << sp->myVal();
```

The `sp->myVal()` call will, when it calls `left->myVal()` will call the `InputExp.myVal()` since `left` contains the address of an `InputExp` object.

Thus inheritance + polymorphism allows classes such as `SumExp` to work with *new* classes such as `InputExp` that may be introduced later in the system.