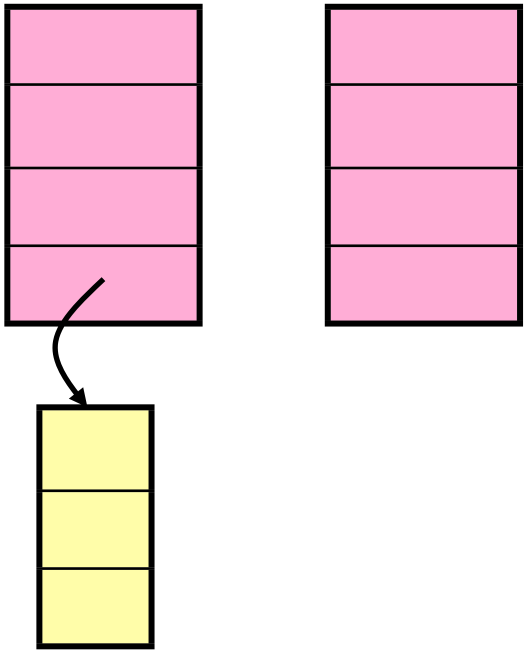
Announcements

MP2 available, due 2/5, 11:59p. EC: 1/29, 11:59p.

```
int main(){
   sphere a, b;
   // change b somehow
   a = b;
   return 0;
}
```

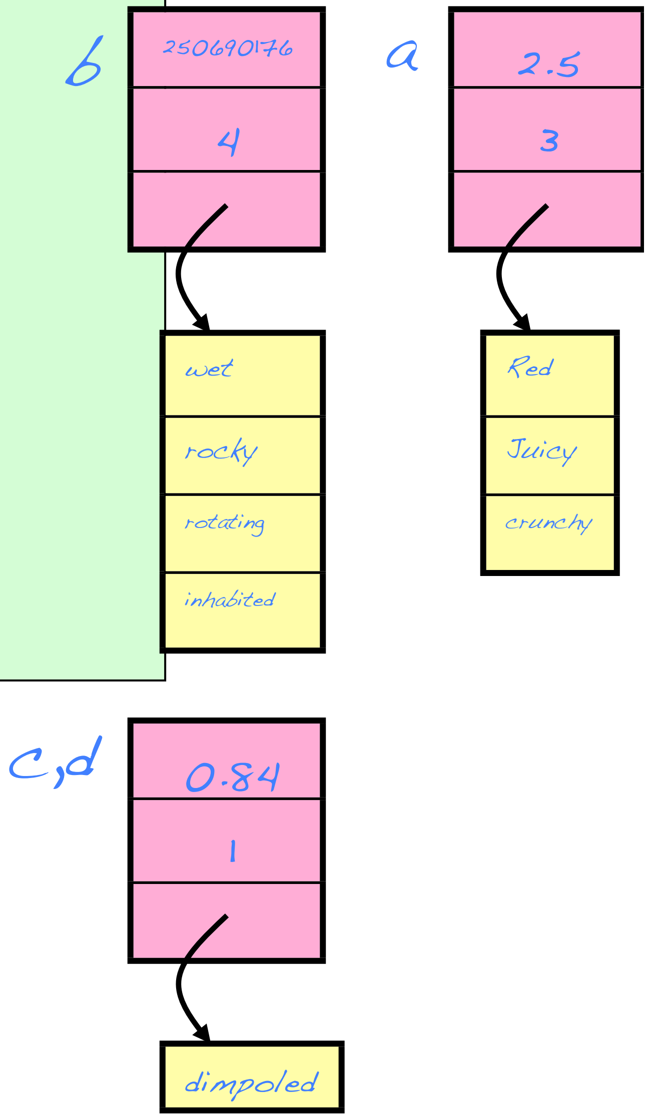


Operator= the plan:

```
clas
publ
sphere
sphere
sphere
~spher

…

priv
double theRadius;
int numAtts;
string * attributes;
};
```

```
…
// overloaded =
_________sphere::operator=(const sphere & rhs){



}
…
```

```
int main(){
   sphere a, b, c;
   // initialize a
   c = b = a;
   return 0;
}
```



Operator=:

```
class sphere{
public:
sphere();
sphere(double r);
sphere(const sphere &
~sphere();

…
private:
double theRadius;
int numAtts;
string * attributes;
};
```

```
…
// overloaded =
sphere & sphere::operator=(const sphere & rhs){
   //protect against re-assignment


      //clear lhs



      //copy rhs




   //return a helpful value

}
…
```

```
int main(){
   sphere a, b;
   // initialize a
   b = a;
   return 0;
}
```

Object Oriented Programming

Three fundamental characteristics:

- encapsulation - separating an object’s data and implementation from its interface.
- inheritance -
- polymorphism - a function can behave differently, depending on the type of the calling object.

Inheritance: a simple first example

```
class sphere {
public:
sphere();
sphere(double r);
double getVolume();
void setRadius(double r);
void display();
private:
double theRadius;
};
```

```
class ball:public sphere {
public:
ball();
ball(double r string n);
string getName();
void setName(string n);
void display();
private:
string name
};
```

inheritance rules:

•

•

•

Protected access: like public to derived classes, like private to anything else

```
class sphere {
public:
    sphere();
    sphere(double r);
    …
    double getVolume();
    void setRadius(double r);
    …
    void display();
private:
    double theRadius;




};
```

```
class ball:public sphere {
public:
    ball();
    ball(double r, string n);
    …
    string getName();
    void setName(string n);
    …
    void display();
private:
    string name;
};
```

```
int main() {
sphere a;
cout << a.surfaceArea;
}
```

Subclass substitution (via examples):

```
sphere s(8.0);
ball b(3.2, “pompom”);

double a = b.getVolume();

void printVolume(sphere t){
   cout << t.getVolume() << endl;}

printVolume(s);
printVolume(b);
```

```
Base b;
Derived d;

b=d;

d=b;
```

```
Base * b;
Derived * d;

b=d;

d=b;
```

something to consider:

```
class sphere {
public:
    sphere();
    sphere(double r);
    …

    void display();
private:
    double theRadius;
};
```

```
class ball:public sphere {
public:
    ball();
    ball(double r string n);
    …
    stri
    void
    .
    void display();
private:
    string name;
};
```

```
void sphere::display() {
   cout << “sphere” << endl;
}
```

```
void ball::display() {
   cout << “ball” << endl;
}
```

ex1

```
sphere s;
ball b;
s.display();
b.display();
```

ex2

```
sphere * sptr;
sptr = &s;
sptr->display();
```

ex3

```
sphere * sptr;
sptr = &b;
sptr->display();
```

“virtual” functions:

```
class sphere {
public:
    sphere();
    sphere(double r);
    …

               void display();
private:
    double theRadius;
};
```

```
void sphere::display() {
   cout << “sphere” << endl;
}
```

```
class ball:public sphere {
public:
    ball();
    ball(double r string n);
    string getName();

    void
    .
               void display();
private:
    string name;
};
```

```
void ball::display() {
   cout << “ball” << endl;
}
```

ex4

```
if (a==0)
   sptr = &s;
else sptr = &b;
sptr->display();
```

virtual functions – the rules:

A virtual method is one a ______________ can override.

A class's virtual methods ________ be implemented. If not, then the class is an "abstract base class" and no objects of that type can be declared.

A derived class is not *required* to override an existing implementation of an ______________ virtual method.

Constructors _____________ be virtual

Destructors can and ____________virtual

Virtual method return type ____________ be overwritten.

Constructors for derived class:

```
ball::ball():sphere()
{
   name = “not known”;
}
```

```
ball b;
```

```
ball::ball(double r, string n):
sphere(r)
{
   name = n;
}
```

```
ball b(0.5,”grape”);
```

“virtual” destructors:

```
class Base{
public:
   Base(){cout<<”Ctor: B"<<endl;}
   ~Base(){cout<<”Dtor: B"<<endl;}
};
class Derived: public Base{
public:
   Derived(){cout<<”Ctor: D"<<endl;}
   ~Derived(){cout<<”Dtor: D"<<endl;}
};
```

```
void main(){
   Base * V = new Derived();
   delete V;
}
```

Abstract Base Classes:

```
class flower {
public:
    flower();
    virtual void drawBlossom() = 0;
    virtual void drawStem() = 0;
    virtual void drawFoliage() = 0;
    …
};
```

```
class daisy:public flower {
public:
    virtual void drawBlossom();
    virtual void drawStem();
    virtual void drawFoliage();
    …
private:
    int blossom; // number of petals
    int stem; // length of stem
    int foliage // leaves per inch
};
```

```
void daisy::drawBlossom() {
// whatever
}
void daisy::drawStem() {
// whatever
}
void daisy::drawFoliage() {
// whatever
}
```

```
flower f;
daisy d;
flower * fptr;
```