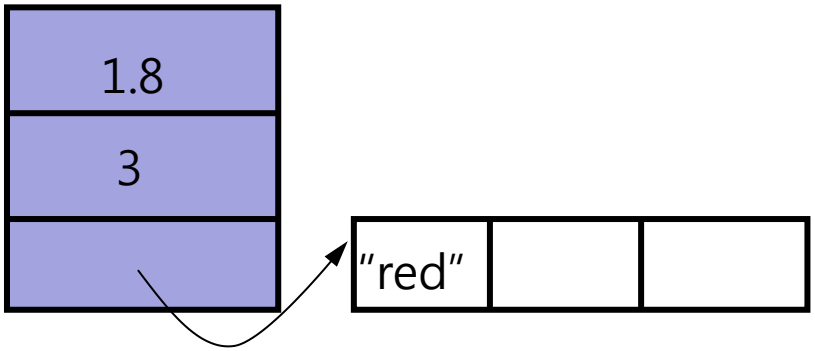
Destructors

```
class sphere {
public:
   sphere();
   sphere(double r);
   sphere(const sphere & orig);
   ~sphere();
   …
private:
   double theRadius;
   int numAtts;
   string * atts;
};
```

```
void myFun(sphere s) {
   sphere t(s);
     // play with s and t
     …

int main() {
     sphere a;
     myFun(a);
     return 0;
}
```

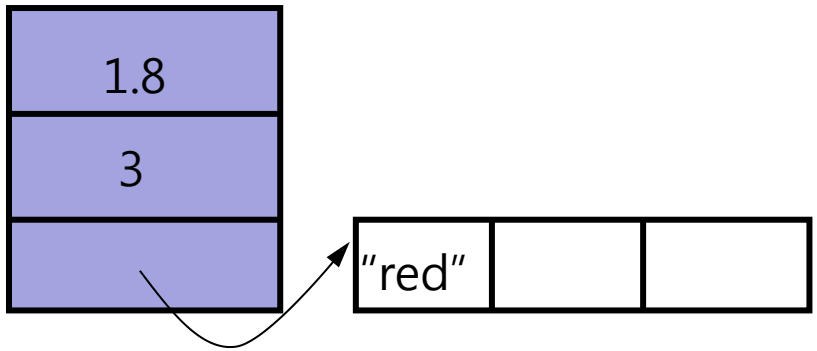



```
// destructor
sphere::~sphere() {

}
```

Destructors

```
class sphere {
public:
   sphere();
   sphere(double r);
   sphere(const sphere & orig);
   ~sphere();
   …
private:
   double theRadius;
   int numAtts;
   string * atts;
};
```

```
int main() {
    sphere * b = new sphere();
    delete b;
    return 0;
}
```




```
// destructor
sphere::~sphere() {


}
```

The destructor - summary

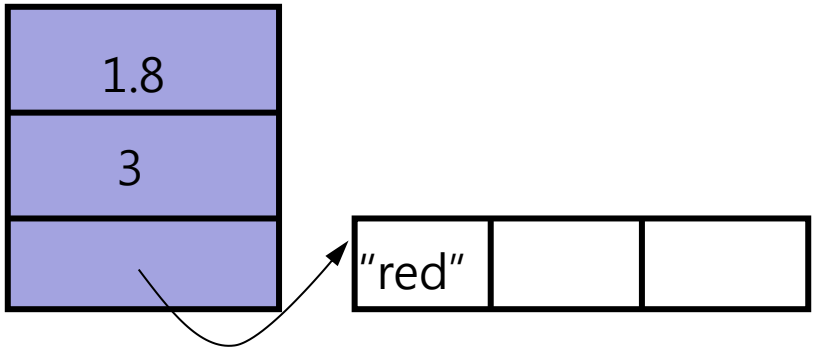
1. Destructor is never "called". Rather, we provide it for the system to use in two situations:

   a) _____(ex. sphere b;)

   b) _____(ex sphere *b = new sphere();
2. If your constructor or _____ allocates dynamic memory, then you need a destructor.

3. Destructor typically consists of a sequence of delate statements.

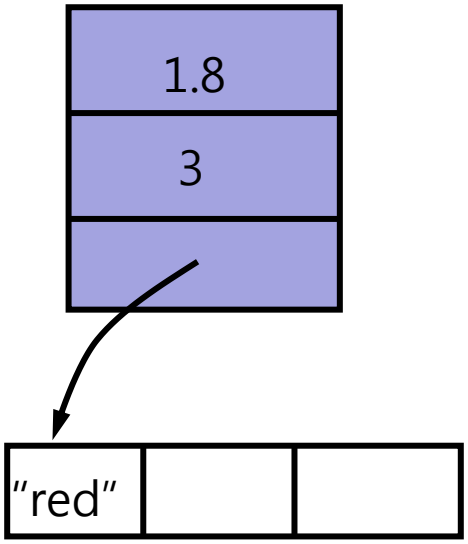
```
class sphere {
public:
   …
   ~sphere();
   …
private:
   double theRadius;
   int numAtts;
   string * atts;
};
```



One more problem:

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

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

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



Overloaded operators:

```
int main() {
   // declare a, b, c



   // initialize a, b



   c = a + b;
   return 0;
}
```

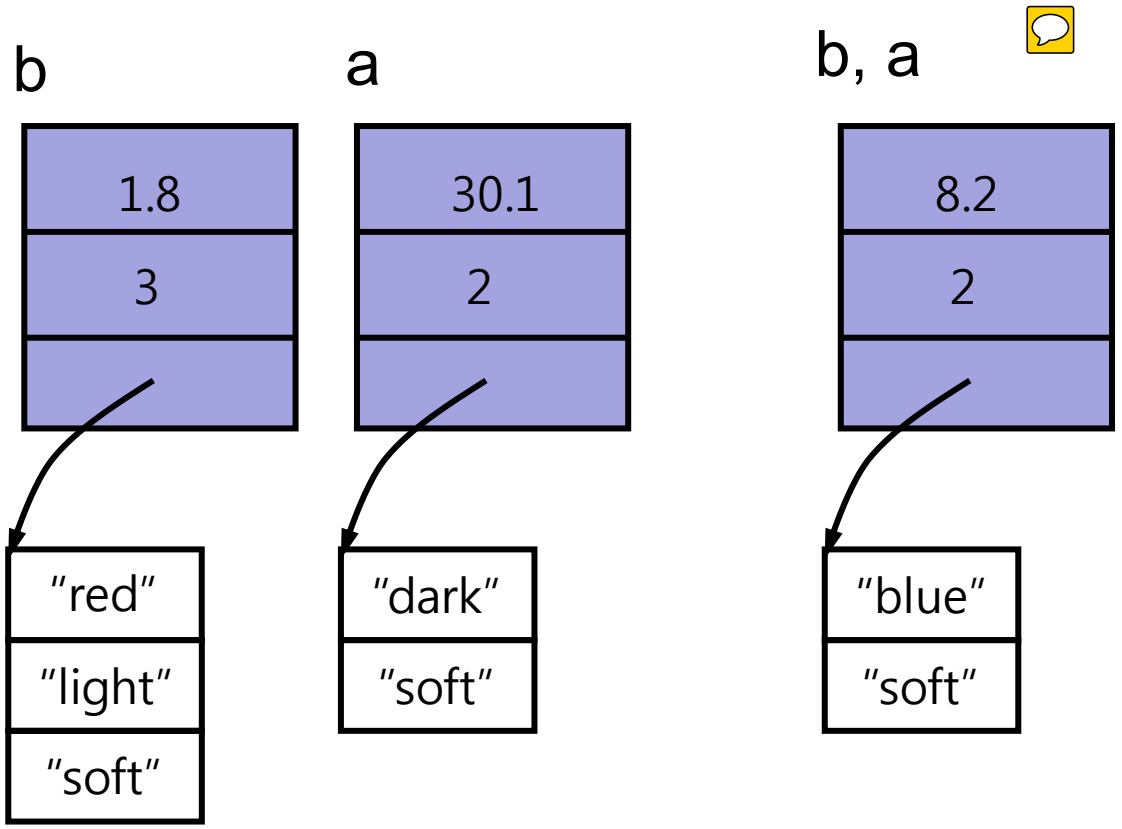
```
// overloaded operator
sphere sphere::operator+ (const
sphere & s) {




}
```

Overloaded operators: what can be overloaded?

Arithmetic operators, logical operators, I/O stream operators

Somethings to think about ….

b = a

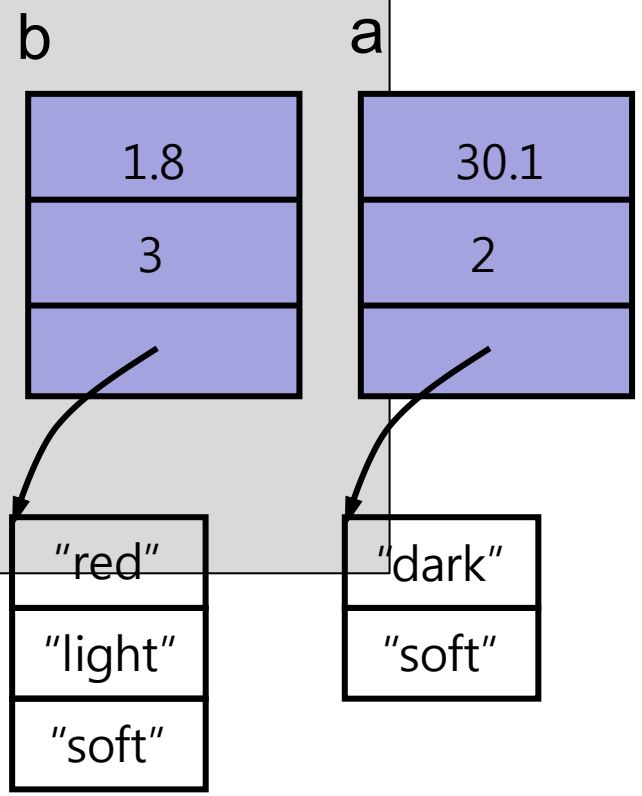


c = b = a

Operator=, the plan:

```
// overloaded =
sphere & sphere::operator= (const sphere & rhs)
{
  // protect against re-assignment
     // clear lhs
        // copy rhs
           // return helpful value
     // return helpful value
}
```

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



Operator=, the plan:

```
class sphere {
public:
  sphere();
  sphere(double r);
  sphere(const sphere &
  ~sphere();
  sphere & operator=(co
sphere & rhs);

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

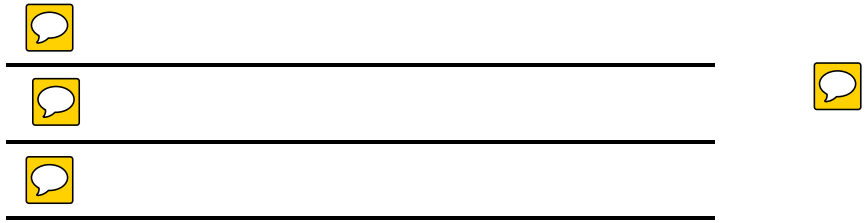
```
// overloaded =
sphere & sphere::operator=
(const sphere & rhs) {
  if (this != &rhs) {
    clear();
    copy(rhs);
  }
  return *this;
}
```

Why not `(*this != rhs)` ?

- 
- 
- 

The Rule of the Big Three:

If you have a reason to implement any one of



then you must implement all three.

Three fundamental characteristics of Object Oriented programming:

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.