Pointer class variables

- When a class has a data member that is a pointer, you have to take extra precautions with that member
  - When the class is destroyed
  - When the class is assigned
  - When the class is copied

Running Example Class

```
class ptrClass
{
public:
    ptrClass( int );
private:
   int x, lenP;
   int * p;
};

ptrClass::ptrClass( int len )
{
   lenP = len;
   p = new int[len];
}
```

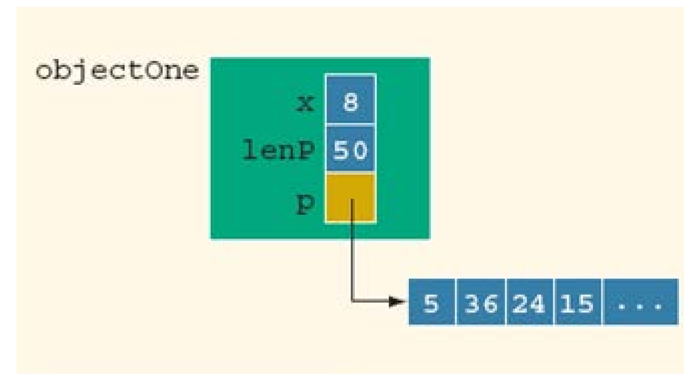

FIGURE 13-22 Object objectOne and its data

Destruction

- Class objects are destroyed when they:
  - Go out of scope
  - Are explicitly deleted
- The memory space of the object (`x`, `lenP`, `p`) is de-allocated
  - But the space `p` points to is not
  - With `p` destroyed, that memory is lost

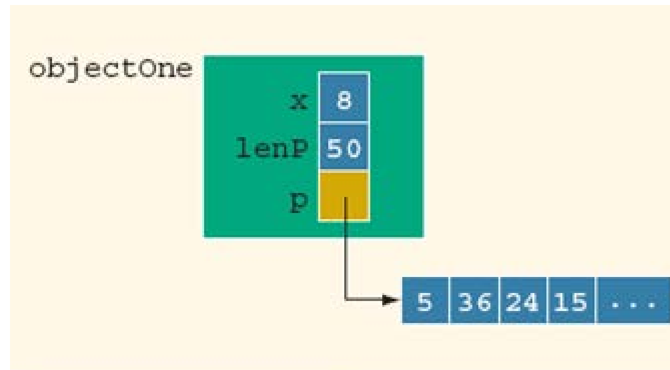

FIGURE 13-22 Object objectOne and its data

Solution: Use the Destructor

- The destructor is always run when the class is destroyed, so make sure to clean up there

```
ptrClass::~ptrClass()
{
  delete [] p;
}
```

- Or, perhaps safer:

```
ptrClass::~ptrClass()
{
  if( p != 0 )
     delete [] p;
}
```

Operator Methods

- When a class object is used as an operand, it invokes a corresponding *operator* method
  - `obj1 + obj2` is the same as `obj1.operator+( obj2 )`
  - `obj1 = obj2` is the same as `obj1.operator=( obj2 )`
  - Operator methods are automatically created for standard operators (e.g. +, -, =)
  - If there is no operator method, you get a compiler error
    - Which is why `cout << obj2` complains about not having a method that works with your class

Operator Overloading

- The class designer can define their own operators
  - This is called *operator overloading*
  - Simply add explicit operator methods to the class
- Common operator method headers, for a class Obj:

```
Obj Obj::operator+( const Obj& rhs );
Obj& Obj::operator=( const Obj& rhs );
```

- Note that the signature must match exactly
  - The expected parameters are constant references
  - The return value is also a reference in some cases

Assignment Issues

- The default assignment operator does a *shallow copy*

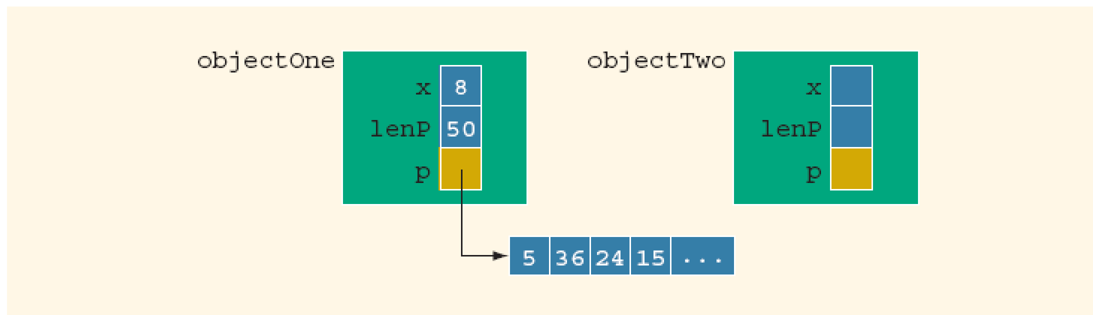

FIGURE 13-23 Objects objectOne and objectTwo

```
objectTwo = objectOne;
```

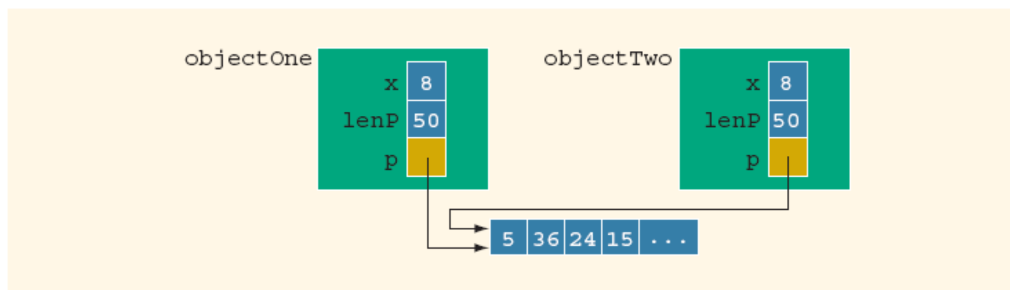

FIGURE 13-24 Objects objectOne and objectTwo after the statement objectTwo = objectOne; executes

Assignment Issues

- Problems with shallow copy:
  - If one object alters the data, it alters for both (this may or may not be what you want)
  - If `objectTwo.p` de-allocates memory space to which it points, `objectOne.p` becomes invalid

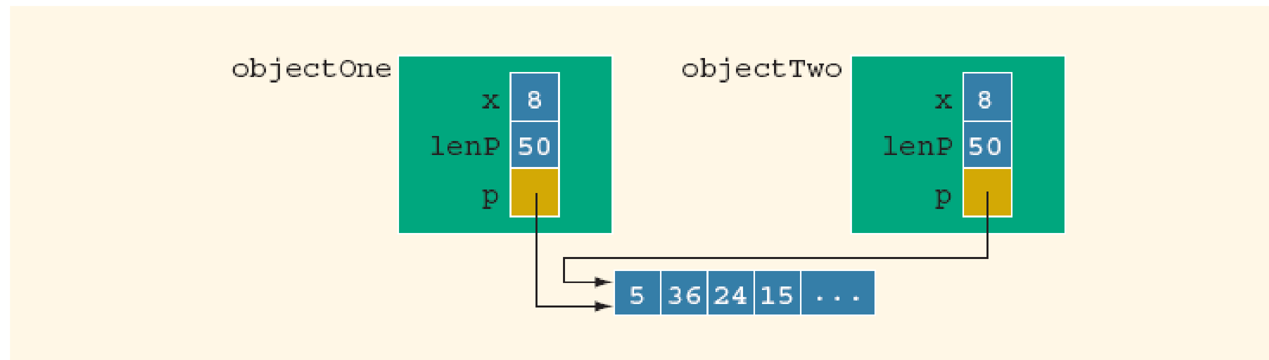

FIGURE 13-24 Objects `objectOne` and `objectTwo` after the statement `objectTwo = objectOne;` executes

Assignment Issues

- Solution: *overload* the assignment operator to do a deep copy
  - Allocate new space
  - Copy data completely (one element at a time in the array case)

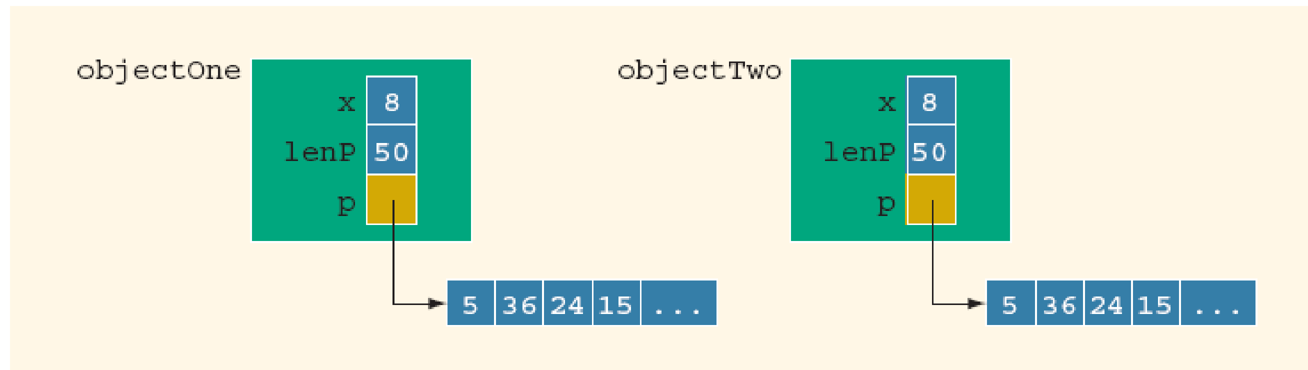

FIGURE 13-25 Objects objectOne and objectTwo

Assignment Operator

- Overload by matching (exactly!) the method signature for *operator=*
  - Takes 1 argument, an object to assign from
    - Same class (ptrClass), by reference, w/ *const* keyword
  - Returns (by reference) itself

```
ptrClass& ptrClass::operator=( const ptrClass& rhs )
{
   delete [] p;
   lenP = rhs.lenP;
   p = new int[lenP];
   for( int i=0;i<lenP;i++ ) {
       p[i] = rhs.p[i];
   }
   return *this;
}
```

Copy Constructor Issues

- Same problem with shallow copy as the assignment operator
- Copy construction happens in three cases:
  - Explicit initialization with another object

```
ptrClass objectThree( objectOne );
```

  - When an object is passed by value
  - When an object is returned by value
- Solution: overload the copy constructor to do a deep copy (just like the assignment operator)
  - Copy constructor takes in an object of the same class

Copy Constructor

- The copy constructor takes an object of the same class as its argument:

```
ptrClass::ptrClass( const ptrClass& to_copy )
{
    lenP = to_copy.lenP;
    p = new int[lenP];
    for( int i=0;i<lenP;i++ ) {
        p[i] = rhs.p[i];
    }
}
```

- The object to copy is passed in:
  - By reference (for efficiency)
  - As a constant (so it does not get changed)

Recap

- When writing a class that has a pointer data member
  – Write a destructor that properly cleans up any dynamically allocated data
  – Overload the assignment operator to do deep copy
  – Include a copy constructor that does deep copy
- These aren’t hard and fast rules, of course
  – There may be reasons to avoid deep copy
  – But they can avoid really common, really difficult to track down problems