Chapter 3

Pointers and Array-Based Lists

Chapter Objectives

- Learn about the pointer data type and pointer variables
- Explore how to declare and manipulate pointer variables
- Learn about the address-of operator and dereferencing operator
- Discover dynamic variables
- Examine how to use the new and delete operators to manipulate dynamic variables

Chapter Objectives (cont'd)

- Learn about pointer arithmetic
- Discover dynamic arrays
- Become aware of shallow and deep copies of data
- Discover the peculiarities of classes with pointer data members
- Explore how dynamic arrays are used to process lists

Pointer Data Types and Pointer Variables

- Pointer variable: variable whose content is a memory address
- Syntax to declare pointer variable:

```
dataType *identifier;
```

- Address-of operator: Ampersand: `&`
- Dereferencing operator: Asterisk: `*`

Pointers

- Statements:

```
int *p;
int num;
```

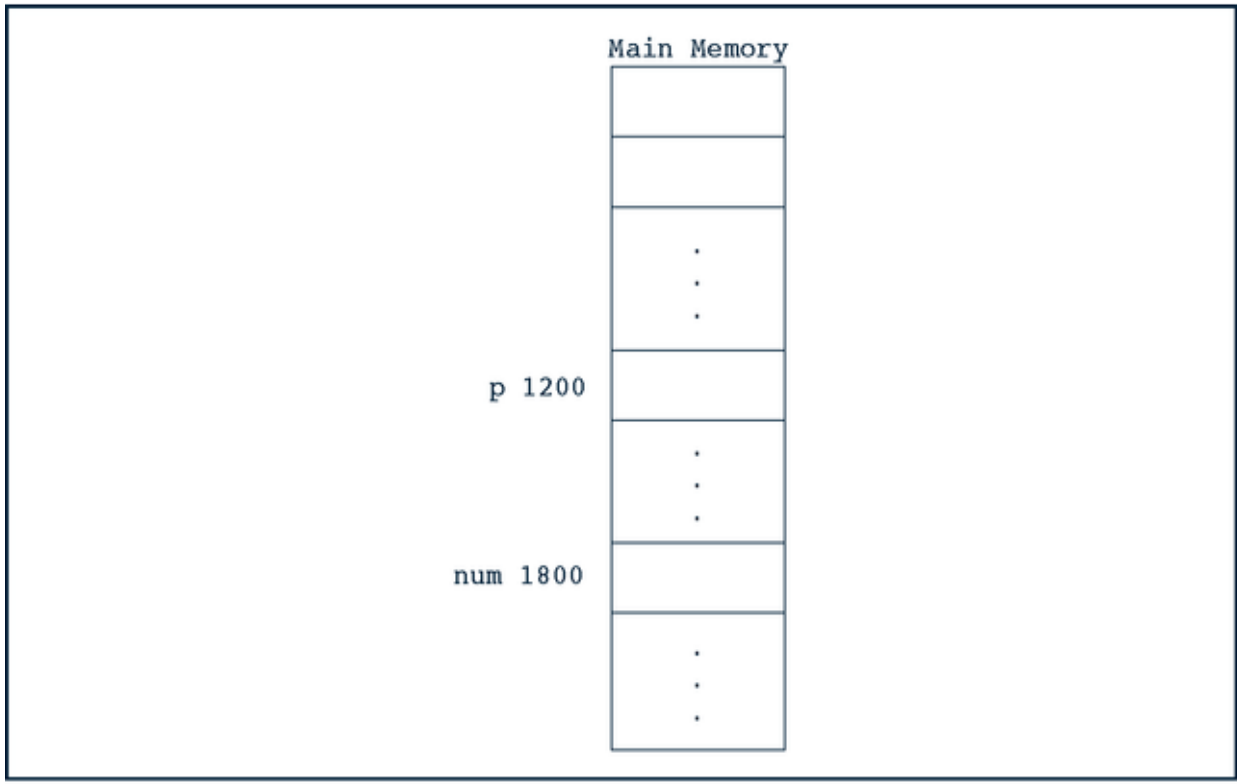

Figure 3-1 Main memory, p, and num

Pointers

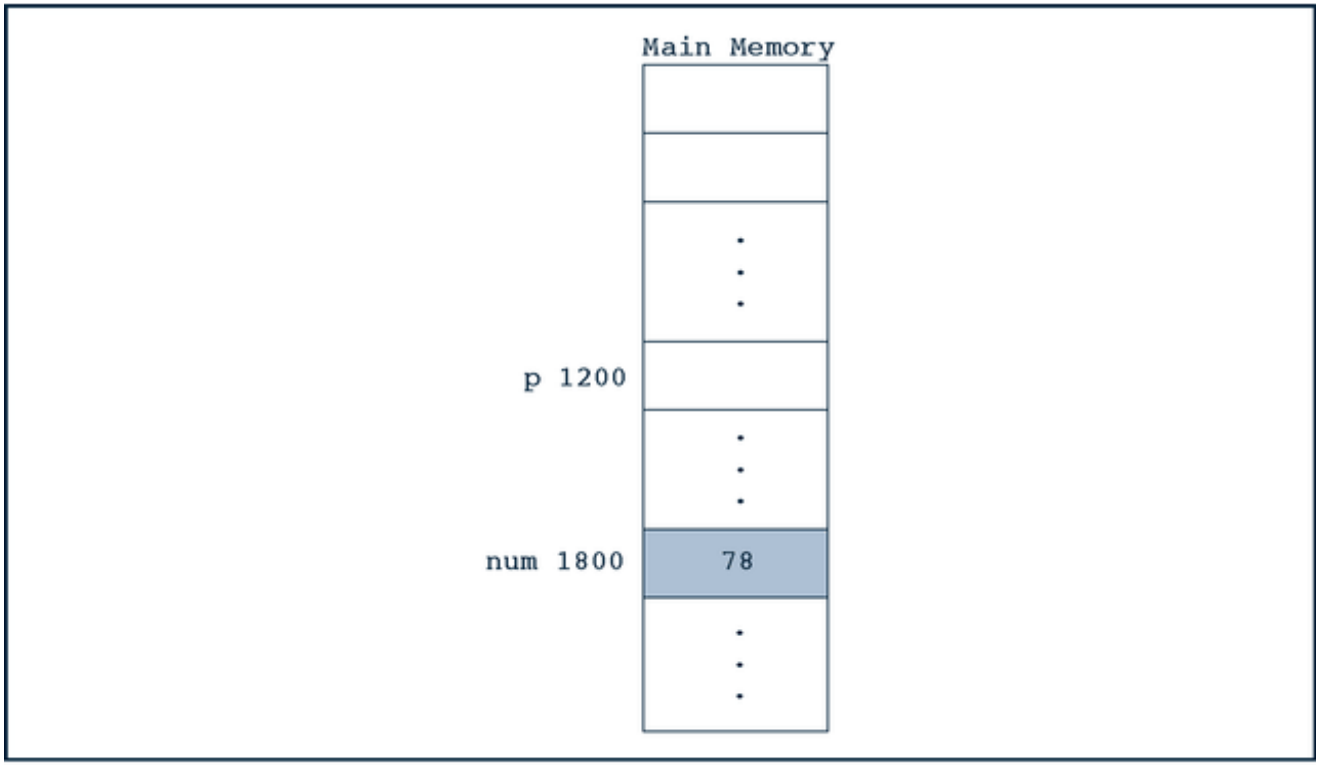

Figure 3-2 num after the statement num = 78; executes

Pointers

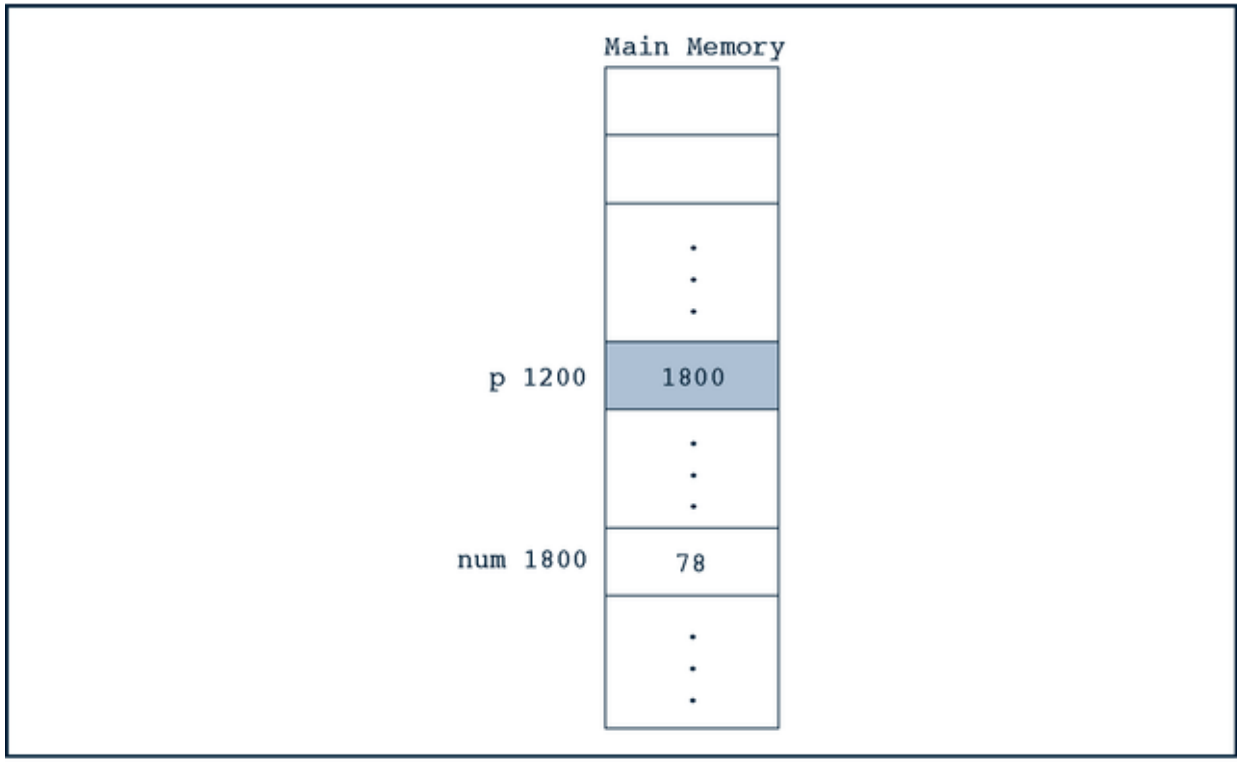

Figure 3-3 p after the statement p = # executes

Pointers

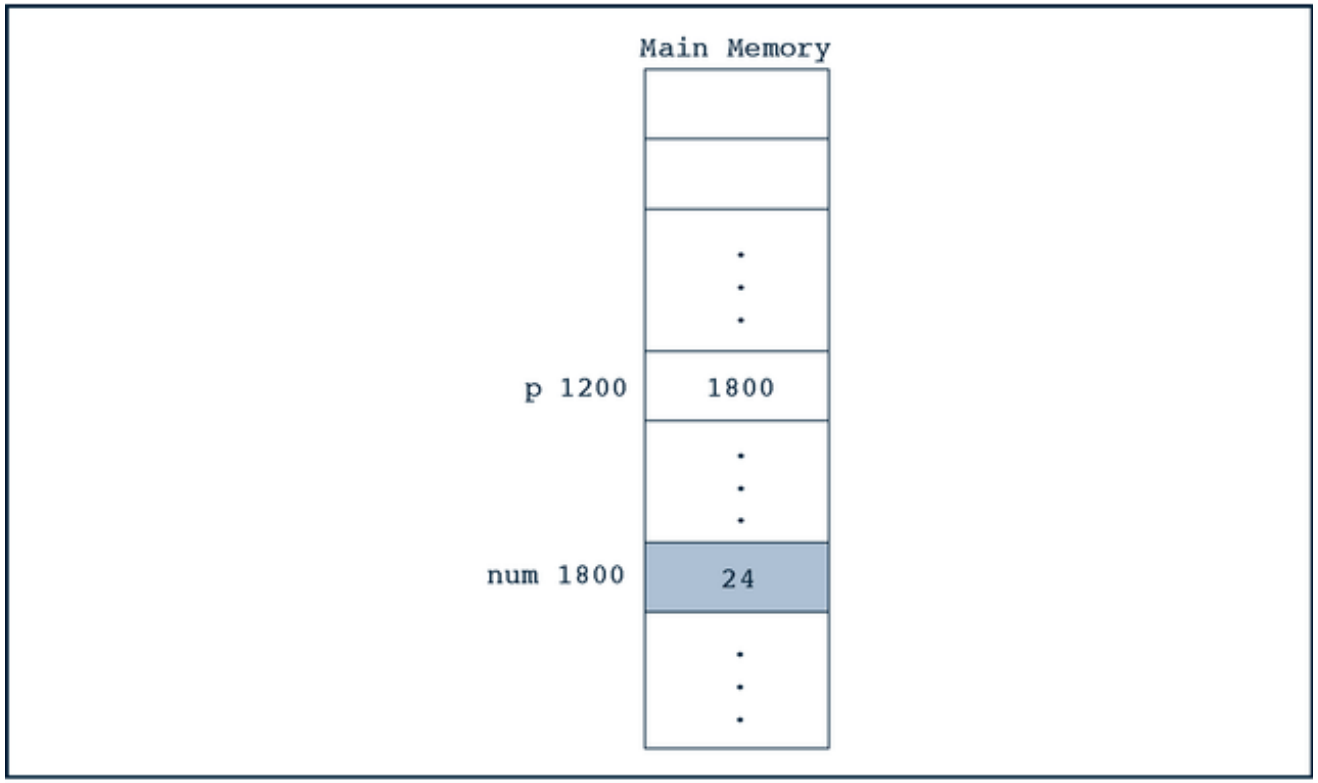

Figure 3-4 *p and num after the statement *p = 24; executes

Pointers

- Summary of preceding diagrams
  - `&p`, `p`, and `*p` all have different meanings
  - `&p` means the address of `p`
  - `p` means the content of `p`
  - `*p` means the content pointed to by `p`, that is pointed to by the content of memory location

Pointers

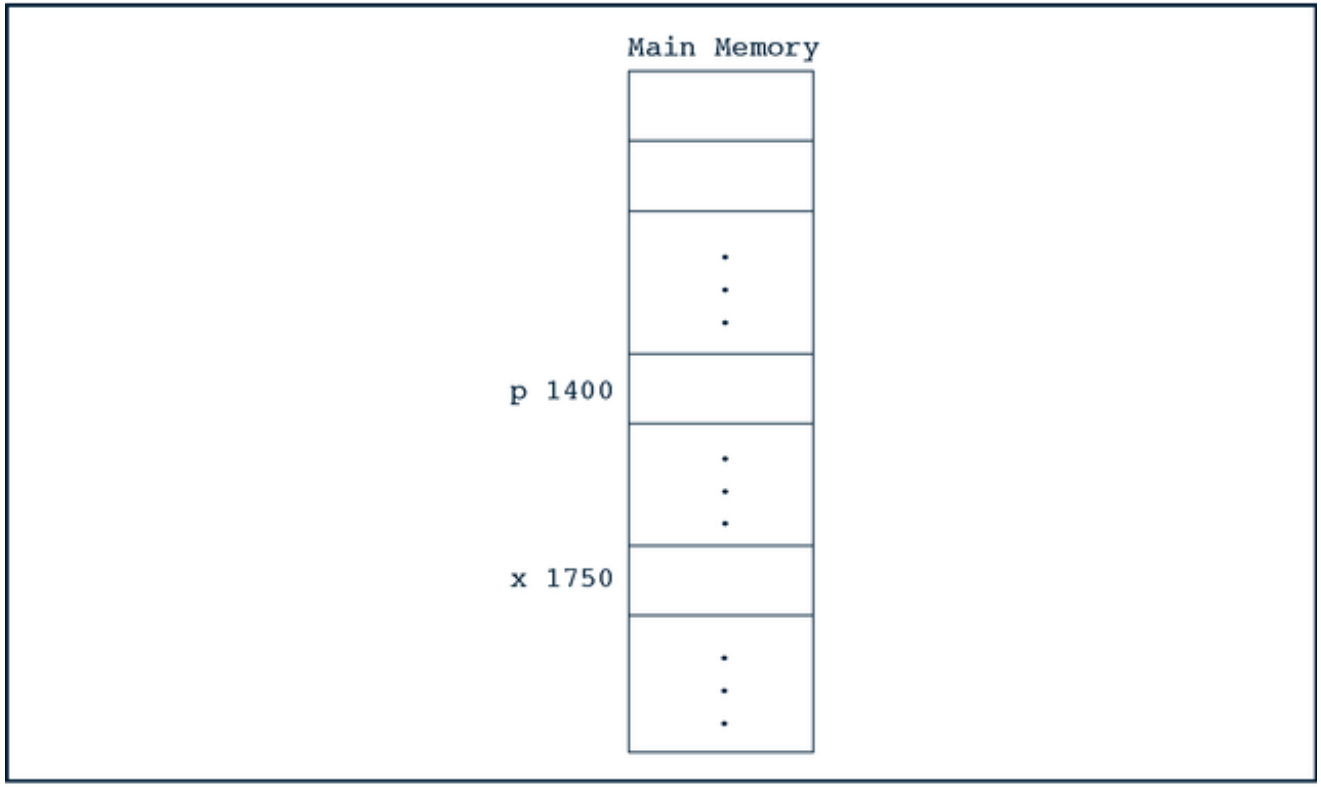

Figure 3-5 Main memory, p, and x

Pointers

```
x = 50;
```

	Value
&p	1400
p	??? (unknown)
*p	Does not exist (undefined)
&x	1750
x	50

```
p = &x;
```

	Value
&p	1400
p	1750
*p	50
&x	1750
x	50

Pointers

```
*p = 38;
```

	Value
&p	1400
p	1750
*p	38
&x	1750
x	38

Classes, structs, and Pointer Variables

The member access operator "`.`" has higher precedence than "`*`", so

```
*studentPtr.gpa = 3.9;
```

only works if `studentPtr` is an instance (not a pointer) and `gpa` is a pointer

To dereference `studentPtr` and access its member `gpa`, use:

```
(*studentPtr).gpa = 3.9;
```

As a shortcut, use the operator "`->`":

```
studentPtr->gpa = 3.9;
```

In general, the syntax for accessing a class (struct) member via its pointer using the operator "`->`" is

```
pointerVariableName->classMemberName
```

Classes, structs, and Pointer Variables

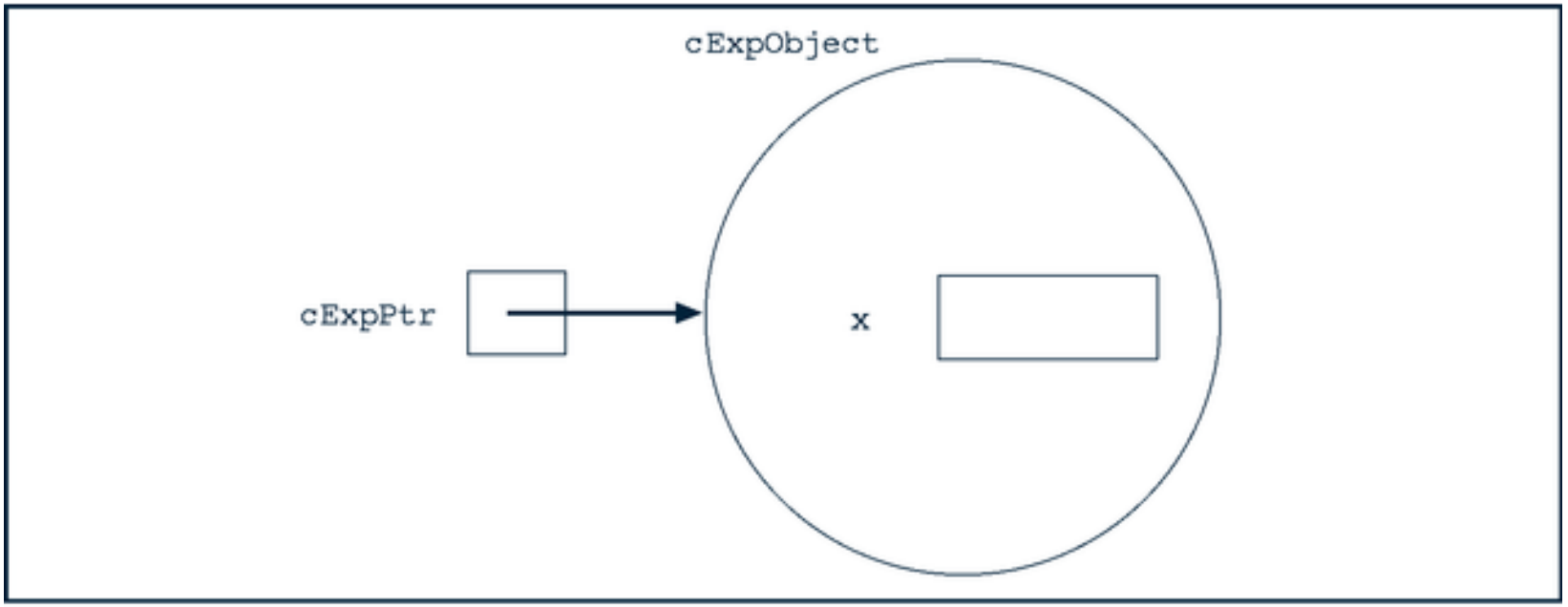

Figure 3-6 `cExpObject` and `cExpPtr` after the statement `cExpPtr = &cExpObject;` executes

Classes, structs, and Pointer Variables

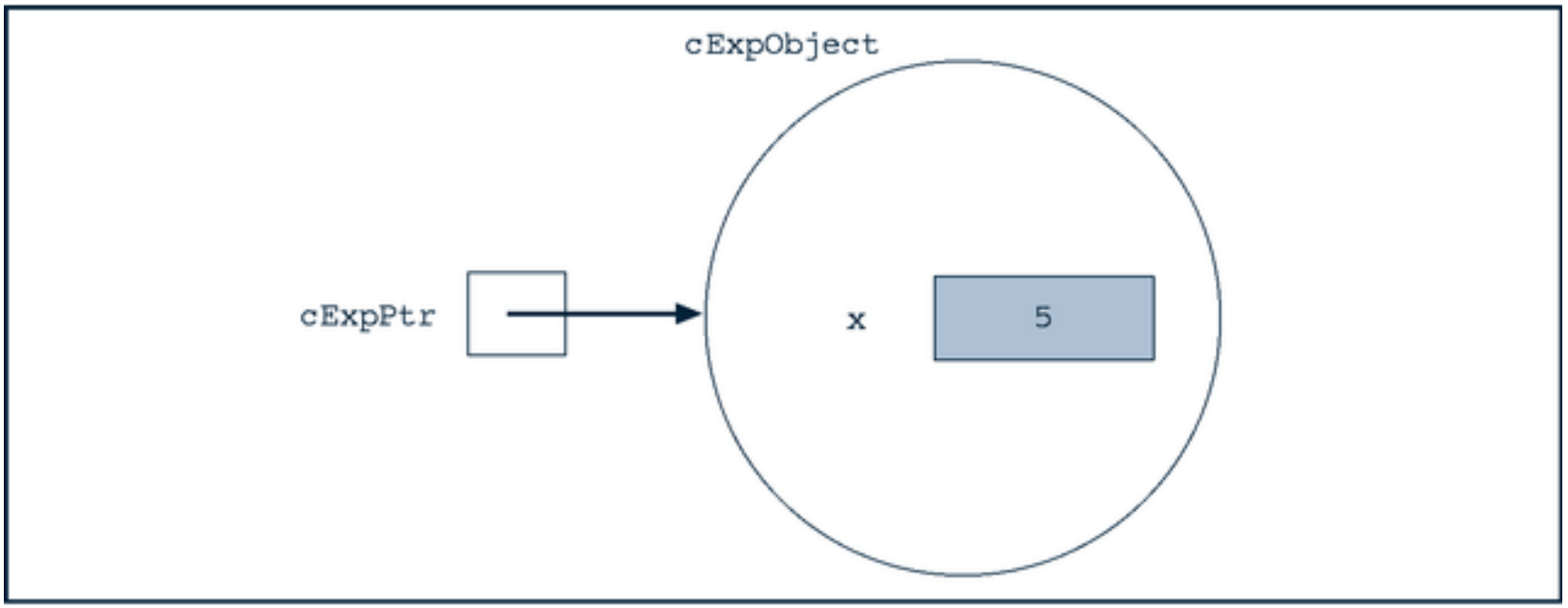

Figure 3-7 `cExpObject` and `cExpPtr` after the statement `cExpPtr ->setX(5);` executes

Dynamic Memory

- Not very interesting when pointers point only to static objects
- The real fun starts when we *dynamically* create objects and point pointers to them
- Allows us to create arrays of custom sizes, minimizing waste and eliminating compile-time constants
- Handled by commands `new` and `delete`

Syntax to use operator new

```
new dataType;              //to allocate a single variable
new dataType[intExp];      //to allocate an array of
                                 // variables
```

```
int *p,*q;
p = new int;        // single int
q = new int[16];    // size-16 array of ints
```

Syntax to use operator delete

```
delete pointer;          //to destroy a single dynamic variable
delete [] pointer;       //to destroy a dynamically created array
```

```
int *p,*q;

p = new int;          // single int

q = new int[16];      // size-16 array of ints

. . .

delete p;          // EVERY new MUST be

delete [] q;       // followed by a delete
```

Operations on pointer variables

```
int *p,*q;
```

- Assignment operations

```
p = q;
```

- Relational operations

```
p == q;     p != q;
```

- Limited arithmetic operations

```
p++; p+=5;  // done in increments of size of type
```

- Array access

```
p[10] = 47;  //only valid after new
```

Functions and Pointers

```
void example(int* &p, double *q)
{
    .
    .
    .
}
```

- `p` is reference parameter (pointer to `int`), `q` is value (pointer to `double`)

Functions Returning Pointers

```
int* testExp(...)
{
    .
    .
    .
}
```

returns a pointer to type `int`.

Shallow Versus Deep Copy and Pointers

```
int *first, *second;

first = new int[10];
```

Figure 3-10 Pointer `first` and the array to which it points

(Assign values to first)

Figure 3-11 Pointer `first` and the array to which it points

Shallow Versus Deep Copy and Pointers

```
second = first;
```

Shallow copy!!!!!

Figure 3-12 `first` and `second` after the statement `second = first;` executes

```
delete [] second;
```

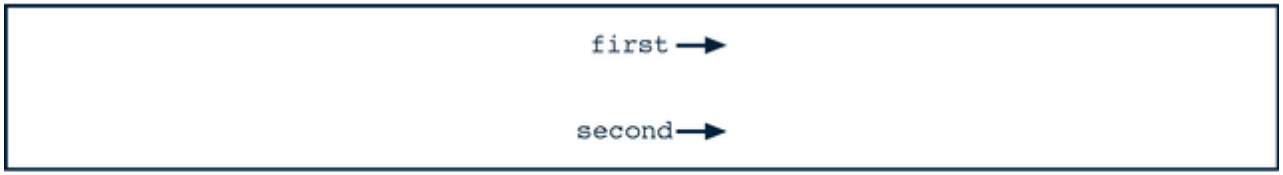

Figure 3-13 `first` and `second` after the statement `delete [] second;` executes

Shallow Versus Deep Copy and Pointers

```
second = new int[10];
for(int j = 0; j < 10; j++)
    second[j] = first[j];
```

Deep copy!!!!!

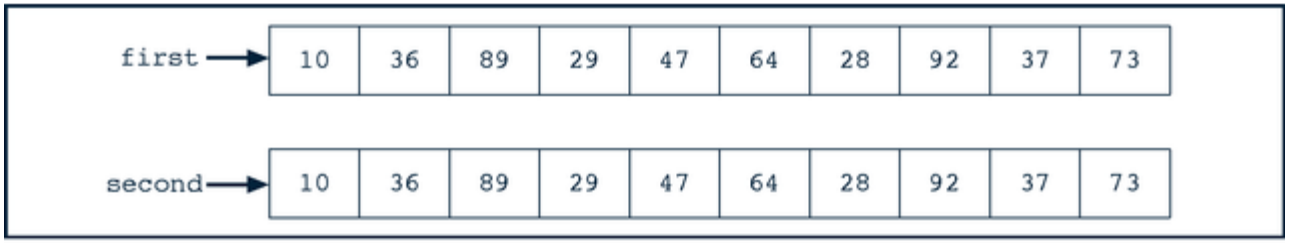

Figure 3-14 `first` and `second` both pointing to their own data

Now, deleting `second` has no effect on `first`

Classes and Pointers

```
class pointerDataClass
{
public:
    ...
private:
    int x;
    int lenP;
    int *p;
};

pointerDataClass objectOne;
pointerDataClass objectTwo;
```

Classes and Pointers

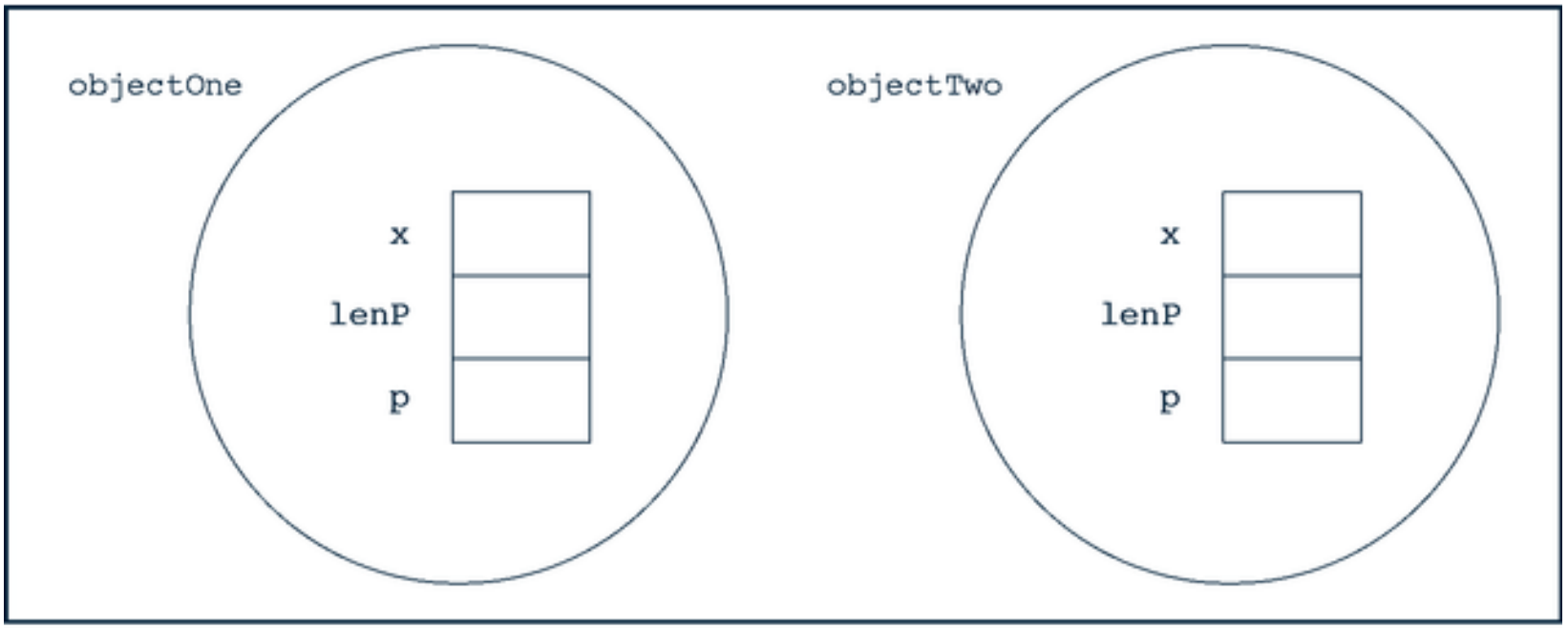

Figure 3-15 Objects `objectOne` and `objectTwo`

Classes and Pointers

```
objectOne.p = new int[objectOne.lenP];
```

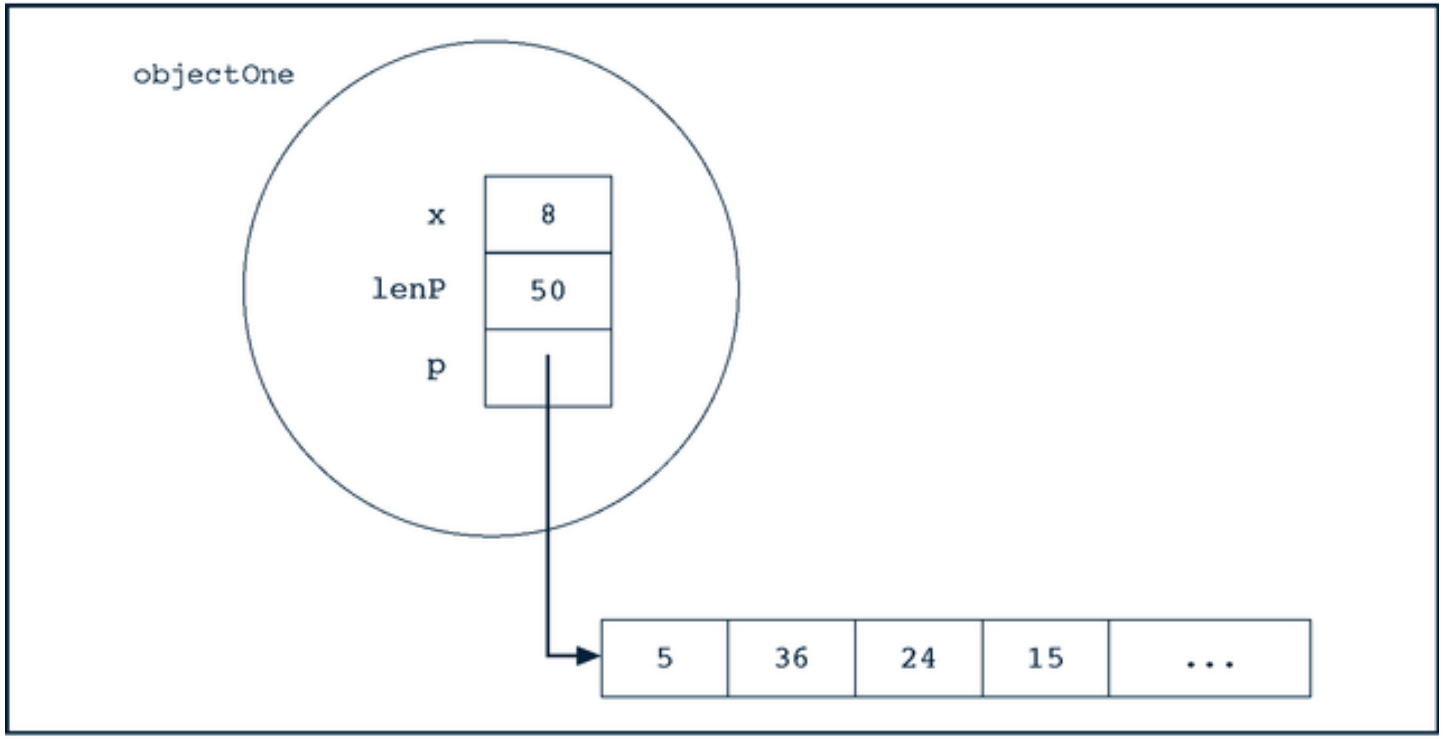

Figure 3-16 Object `objectOne` and its data

Classes and Pointers

- What happens when objectOne goes out of scope?
- What happens to the array pointed to by p?
  - Memory leak!
- Need to free up this memory before the object disappears (preferably in the *destructor*)
- Again, EVERY new MUST be followed by a delete!

Freeing Memory in the Destructor

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

class pointerDataClass
{
public:
    ~pointerDataClass();
...
private:
    int x;
    int lenP;
    int *p;
};
```

Assignment Operator

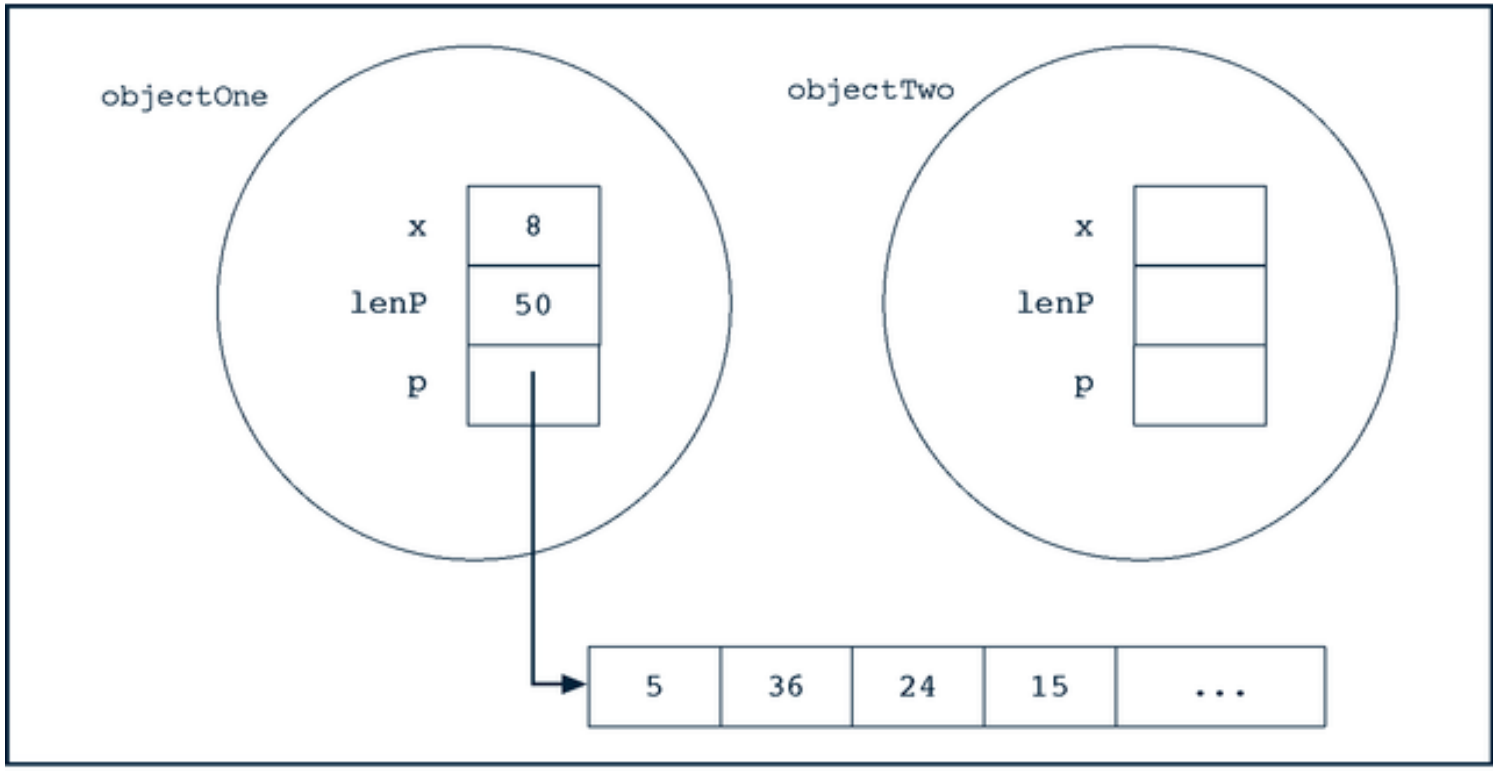

Figure 3-17 Objects `objectOne` and `objectTwo`

Assignment Operator (cont'd)

```
objectTwo = objectOne;    //shallow copy
```

- The destructor of one deletes the memory for `p` of both!

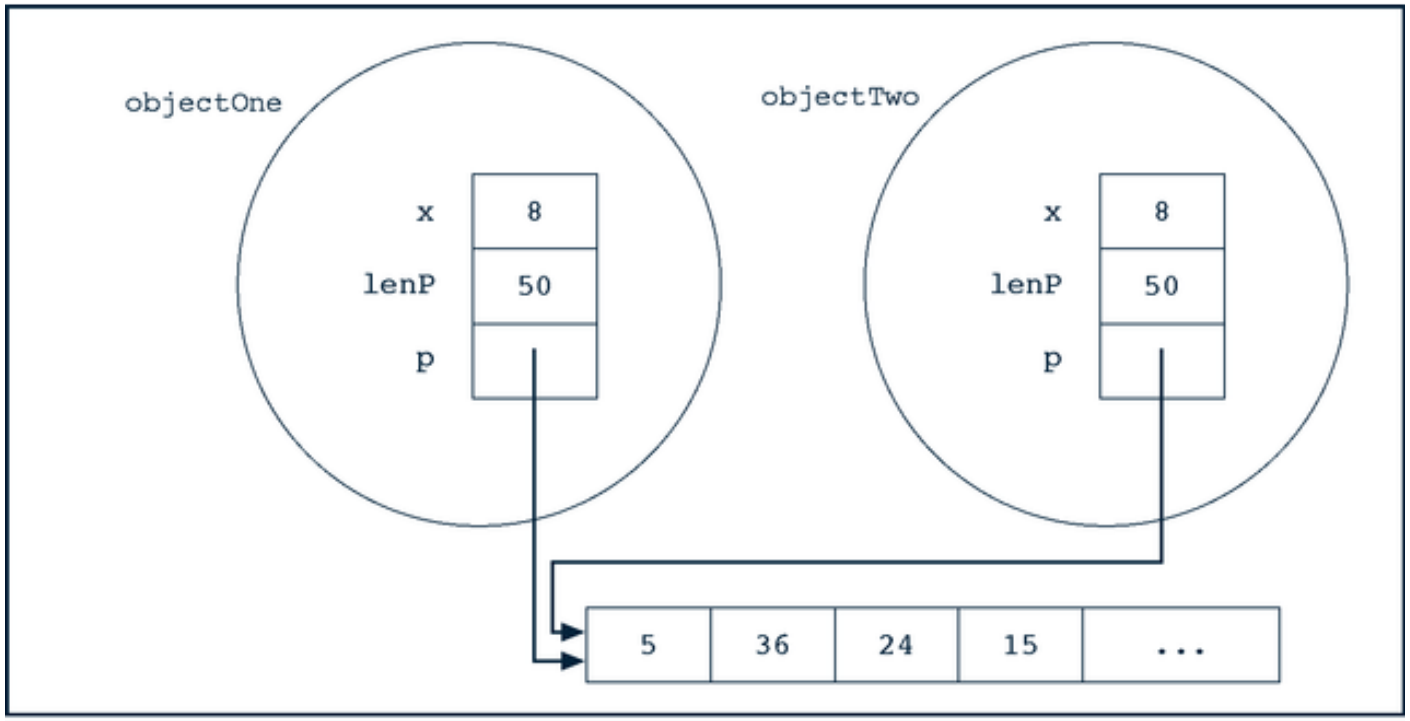

Figure 3-18 Objects `objectOne` and `objectTwo` after the statement `objectTwo = objectOne;` executes

Assignment Operator (cont'd)

How can we make `objectTwo = objectOne;` a deep copy?

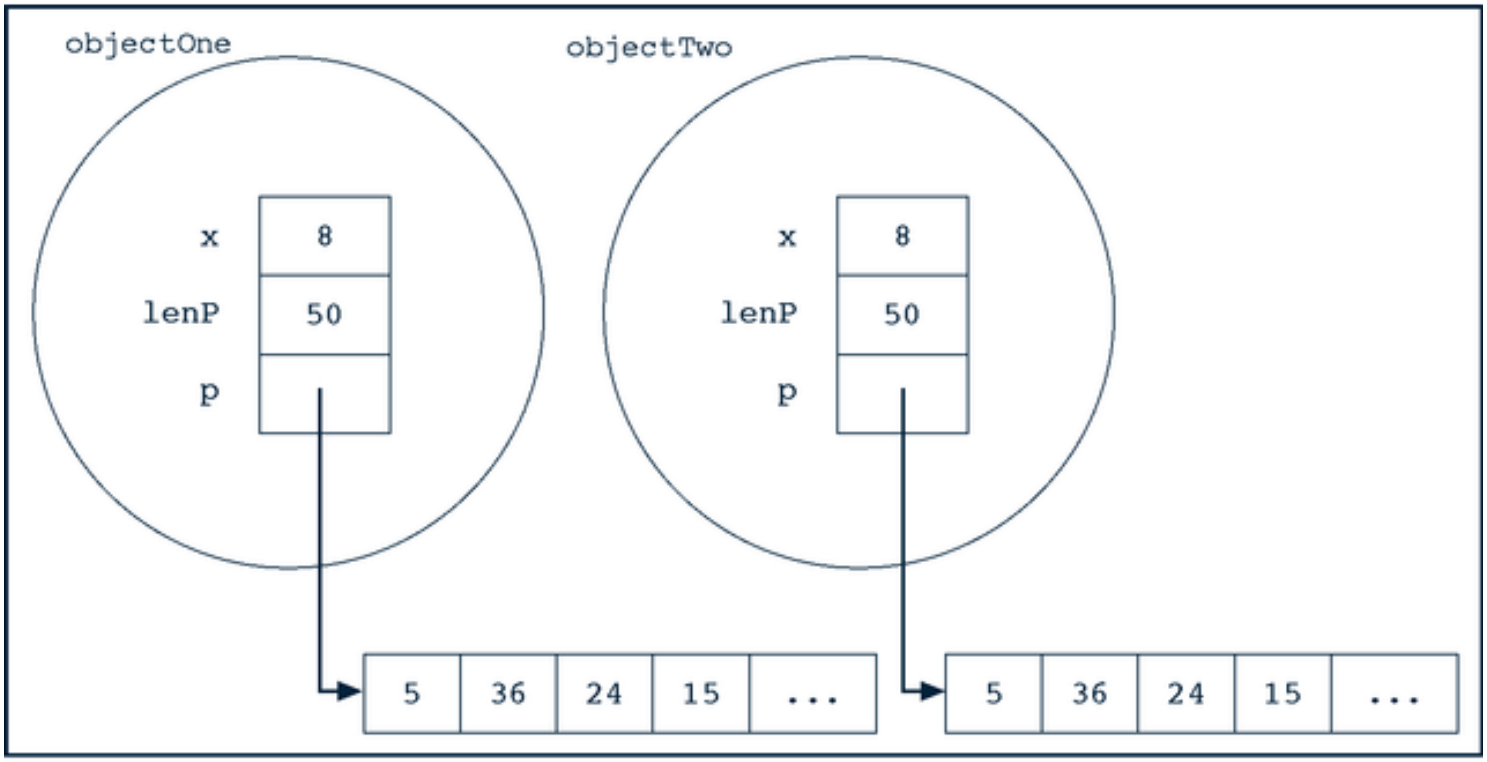

Figure 3-19 Objects `objectOne` and `objectTwo`

Overloading Assignment Operator

Function Prototype (to be included in the definition of the class):

```
const className& operator=(const className&);
```

Function Definition:

```
const className& className::operator=(const className& rightObject)
{
        //local declaration, if any
        if(this != &rightObject)  //avoid self-assignment
       {
         x = rightObject.x;
         lenP = rightObject.lenP;
         if (p != NULL) destroyList();
         if (lenP > 0) {
             p = new int[lenP];
             for (int i=0; i<lenP; i++) p[i] = rightObject.p[i];
           }
       }
         //return the object assigned
       return *this;
}
```

Overloading the Assignment Operator

- Definition of function `operator=`
  - Only one formal parameter
  - Formal parameter generally const reference to particular class
  - Return type of function is reference to particular class

Copy Constructor

• Built-in constructor called when an object of same type used as parameter

```
pointerDataClass objectThree(objectOne);
                    //shallow copy
```

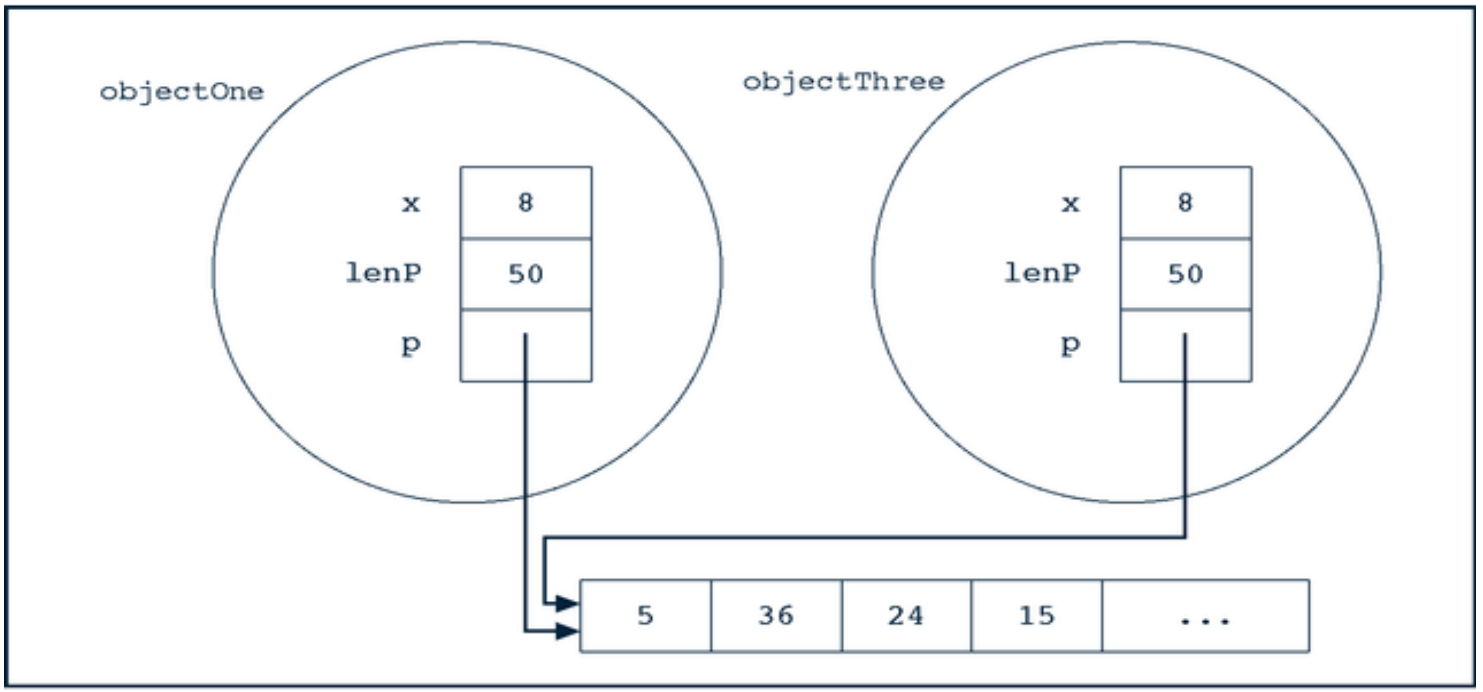

Figure 3-20 Objects `objectOne` and `objectThree`

Copy Constructor

- The copy constructor automatically executes in the following situations
  - When an object is declared and initialized by using the value of another object
  - When, as a parameter, an object is passed by value
  - When the return value of a function is an object

Copy Constructor

- If a class has pointer data members:
  - During object declaration, the initialization of one object using the value of another object would lead to a shallow copying of the data if the default memberwise copying of data is allowed
  - If, as a parameter, an object is passed by value and the default member-wise copying of data is allowed, it would lead to a shallow copying of the data

Copy Constructor

General syntax to include the copy constructor in the definition of a class:

```
className(const className& otherObject);
```

Function definition similar to overloaded assignment operator

Classes with Pointer Data Members

- The “Big 3” things to remember to do:
  1. Include destructor in the class
  2. Overload assignment operator for class
  3. Include copy constructor

Overloading Index Operator ([ ])

Syntax to declare the operator function operator [ ] as a member of a class for nonconstant arrays:

`Type& operator[](int index);`

```
Class classTest {
. . .
private:
Type *list;
int arraySize;
};

Type& classTest::operator[](int index) {
  assert(0 <= index && index < arraySize);
  return(list[index]);
}
```

Chapter Summary

- Pointer data types and variables
- Dynamic variables
- Pointer arithmetic
- Dynamic arrays
- Shallow and deep copying
- Peculiarities of classes with pointer data members
- Processing lists