SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Pointers and Dynamic Arrays

Learning Objectives

- Trace through code with simple pointers that contain the addresses of individual variables
- Use pointer variables along with the C++ `new` operator to allocate single dynamic variables and dynamic arrays
- Use the C++ `delete` operator to release dynamic variables and dynamic arrays when they are no longer needed
- Follow the behavior of pointers and arrays as parameters to functions
- Implement container classes so that the elements are stored in a dynamic array with a capacity that is adjusted by the class's member functions as needed

Introduction

- The container classes' capacity is declared as a constant in the class definition (bag::CAPACITY)
  - If we need bigger bags, then we can increase the constant and recompile the code
  - All the bags will be of the same size

- What if a program needs one large bag and many small bags?

Introduction

- Solution:
  - Provide control over the size of each bag, independent of the other bags
  - This control can come from **dynamic arrays**

- Dynamic arrays
  - Arrays whose size is determined while a program is actually running (not at compile time)

POINTERS AND DYNAMIC MEMORY

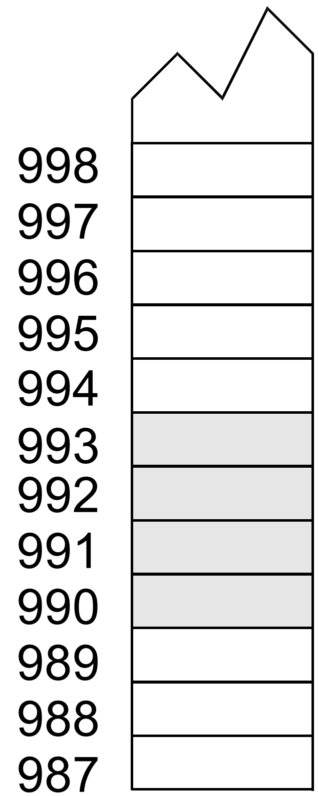
Pointers

- ***Pointer***: the memory address of a variable

- Memory addresses: numbers labeling each byte
  - When a variable occupies several adjacent bytes, the memory address of the first byte is the address of the variable

An integer variable might require four bytes of memory.

A program might provide these four bytes for an integer `i`.



Pointer Variables

- Pointer variable must be declared by placing an asterisk before the variable name

```
Type_Name *var_name1;
```

The type of data that the pointer variable can point to

Asterisk

The name of the pointer variable

- Example

```
double *my_first_ptr;

char *c1_ptr, *c2_ptr;
```

Pointer Variables (cont.)

- Assignment statement

```
int *example_ptr;
int i;

example_ptr = &i;
```

- & operator
  - address operator
  - provides the address of a variable

- &i: the address of the integer variable i

Pointer Variables (Cont'd)

- *example_ptr: the variable pointed to by example_ptr
- **Dereferencing operator**

```
int *example_ptr;
int i;

i = 42;
example_ptr = &i;

cout << &i << endl;
cout << example_ptr << endl;

cout << i << endl;
cout << *example_ptr << endl;

*example_ptr = 0;
cout << i << endl;
cout << *example_ptr << endl;
```

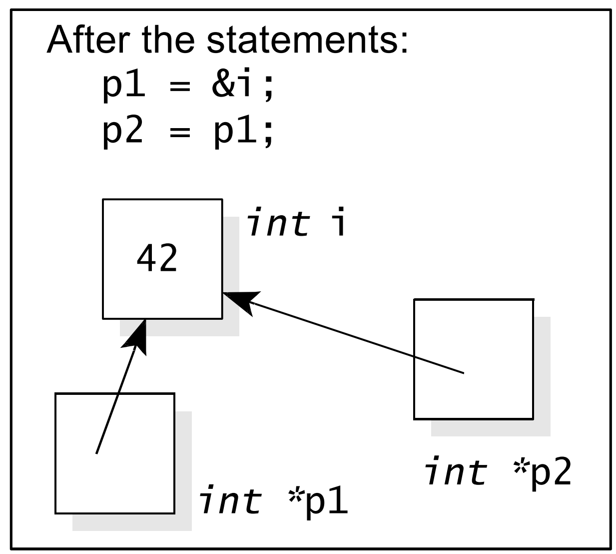
Using the Assignment Operator with Pointers

- You can copy the value of one pointer variable to another with the usual assignment operator

```
int i = 42;
int *p1;
int *p2;

p1 = &i;
p2 = p1;

cout << *p1 << endl;
cout << *p2 << endl;
```

p1 now points to i

p2 also points to i



Using the Assignment Operator with Pointers (cont.)

- There is a critical distinction between a pointer variable (such as p1) and the thing it points to (such as *p1)

```
p2 = p1                                *p2 = *p1
```

What is the difference?

Dynamic Variables and the `new` Operator

- **Dynamic variables** are like ordinary variables, with two important differences:
- ✓ **Not declared**
- ✓ **Created during the execution of a program**
  - C++ programs use *new*

- The creation of new dynamic variables is called **memory allocation**

```
double *d_ptr;  d_ptr = new double;
```

Using new to Allocate Dynamic Arrays

- *new* can allocate an entire array at once
- returns a pointer to the first component of the array

```
double *d_ptr;
d_ptr = new double[10];
d_ptr[9] = 3.14;
```

- Dynamic variable for a class

```
throttle *t_ptr;
t_ptr = new throttle(5);
```

The Heap and the bad_alloc Exception

- When *new* allocates a dynamic variable or dynamic array, the memory comes from a location called the program's **heap (free store)**
- When the heap runs out of room, the *new* operator fails
- The *new* operator usually indicates failure by throwing an exception called the *bad_alloc* exception
- Normally, an exception causes an error message to be printed and the program to halt

The `delete` Operator

- The size of the heap varies from one computer to another, it could be just a few thousand bytes or more than a billion
- Even with small programs, it is an efficient practice to release any heap memory that is no longer needed
- The delete operator is used to return the memory of a dynamic variable back to the heap where it can be reused for more dynamic variables

- Example

```
int *example_ptr;
example_ptr = new int;
…
delete example_ptr;
```

The delete Operator (Cont'd)

- delete operator can also free a dynamic array of components
- To free an entire array, the array brackets [ ] are placed after the word delete

- Example

```
int *example_ptr;
example_ptr = new int[50];
…

delete [ ] example_ptr;
```

Define Pointer Types

- `typedef int* int_pointer;`
  - A type definition usually appears in a header file or with the collection of function prototypes that precede a main program

```
int_pointer i_ptr;        // int *i_ptr;
```

Exercises

Write code that allocates a new array of 1000 integers; places the numbers 1 through 1000 in the components of the new array; and returns the array to the heap.

Exercises

```
int *p1;
int *p2;

p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << " and " << *p2 << endl;
delete p1;
p1 = p2;
cout << *p1 << " and " << *p2 << endl;
*p1 = 300;
cout << *p1 << " and " << *p2 << endl;
*p2 = 400;
cout << *p1 << " and " << *p2 << endl;
delete p1;
```

POINTERS AND ARRAYS AS PARAMETERS

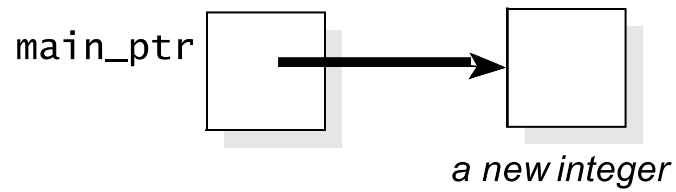
Value Parameters that are Pointers

```
void make_it_42(int* i_ptr)
{
// Precondition: i_ptr is pointing to an integer variable.
// Postcondition: The integer that i_ptr is pointing at has
// been changed to 42.

    *i_ptr = 42;

}
```

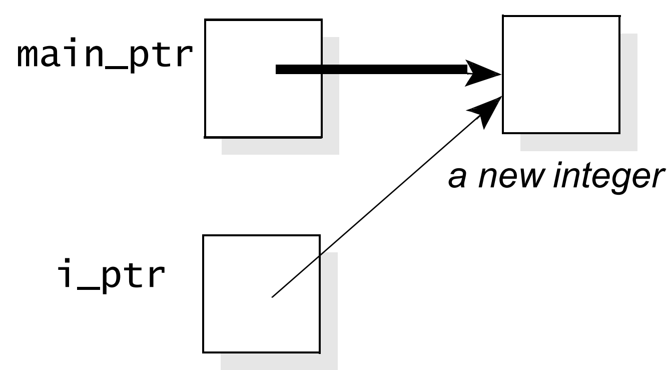
- Parameter i_ptr has type int*, a pointer to an integer
- A value parameter because the reference symbol & does not appear
- Note: The body of the function does not actually change i_ptr; it changes only the integer that i_ptr points to

Value Parameters that are Pointers (cont.)

```
int *main_ptr;
main_ptr = new int;
```

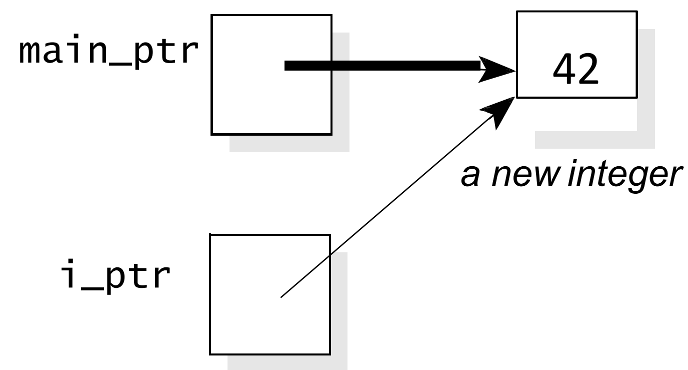



```
make_it_42(main_ptr);
```



Value Parameters that are Pointers (cont.)

```
void make_it_42(int* i_ptr)
{
      *i_ptr = 42;
}
```



- When the function returns, the formal parameter i_ptr is no longer available
- However, the pointer variable main_ptr is still around, and it is still pointing to the same location, but the location has a new value of 42

Array Parameters

- When a parameter is an array, it is automatically treated as a pointer that points to the first element of the array
- If the body of the function changes the components of the array, the changes do affect the actual argument

```
void make_it_all_42(double data[ ], size_t n)
// Precondition: data is an array with at least n
// components.
// Postcondition: The first n elements of the data have
// been set to 42.
{
    size_t i;
    for (i = 0; i < n; ++i)
            data[i] = 42;
}
```

Array Parameters (Cont.)

```
double main_array[10];
make_it_all_42(main_array, 10);
cout << main_array[5];
```

- The actual argument of `make_it_all_42` may be a dynamic array:

Data allocated on heap memory

```
double *numbers;
numbers = new double[10];
make_it_all_42(numbers, 10);
```

Set all elements to 42

const Parameters that are Pointers or Arrays

- A parameter that is a pointer may also include the const keyword
- The functions may examine the item that is pointed to, but changing the item (or array) is forbidden

```
bool is_3(const int* i_ptr);

double average(const double data[ ], size_t n);
```

const Parameters that are Pointers or Arrays (Cont'd)

```
bool is_3(const int* i_ptr)
{
   // Precondition: i_ptr is pointing to an integer variable.
   // Postcondition: The return value is true if *i_ptr is 3.
   return (*i_ptr == 3);
}

double average(const double data[ ], size_t n)
// Library facilities used: cassert, cstdlib
{
   size_t i;    // An array index
   double sum;  // The sum of data[0] through data[n - 1]

   assert(n > 0);

   // Add up the n numbers and return the average.
   sum = 0;

   for (i = 0; i < n; ++i)
     sum += data[i];

   return (sum/n);
}
```

Reference Parameters that are Pointers

- A reference parameter that is a pointer is used when a function:

1. Changes a pointer parameter so that the pointer points to a new location
2. The programmer needs the change to affect the actual argument

Reference Parameters that are Pointers (cont.)

- p is a pointer to a double (double*) and it is a reference parameter (indicated by the symbol &)

```
void allocate_doubles(double*& p, size_t& n)
// Postcondition: The user has been prompted for a size n, and this
// size has been read.
// The pointer p has been set to point to a new dynamic array
// containing n doubles.
// NOTE: If there is insufficient dynamic memory, then bad_alloc is
// thrown.
{
    cout << "How many doubles should I allocate?" << endl;
    cout << "Please type a positive integer answer: ";
    cin >> n;
    p = new double[n];
}
```

Reference Parameters that are Pointers (cont.)

- In a program, we can use *allocate_doubles* to allocate an array of double values, with the size of the array determined by interacting with the user

```
double *numbers;
size_t array_size;
allocate_doubles(numbers, array_size);
```

Reference Parameters that are Pointers (cont.)

- Define a type definition (typedef) for a pointer type to avoid the cumbersome syntax of *&
- Define *double_ptr* to be a pointer to a double number:

```
typedef double* double_ptr;
```

```
void allocate_doubles(double_ptr& p, size_t& n);
```

Reference Parameters that are Pointers (cont.)

```
#include <iostream>

void function_a(int *& a)
{
    *a += 5;
    int* c = new int(7);
    a = c;
}

void function_b(int * a)
{
    *a += 5;
    int* c = new int(7);
    a = c;
}
```

```
int main()
{
    int* myInt = new int(5);
    int* myInt2 = new int(5);

    function_a(myInt);
    std::cout << myInt << std::endl;
    std::cout << *myInt << std::endl;

    function_b(myInt2);
    Std::cout << myInt2 << std::endl;
    Std::cout << *myInt2 << std::endl;

    return 0;
}
```

```
Output:
0x100202210
7
0x100200570
10
```

Overview of Parameter Types

- Value Parameter
```
void function(double p);
```
- Reference Parameter
```
void function(double& p);
```
- const Reference Parameter
```
void function(const double& p);
```
- Value Parameter that is Pointer
```
void function(double* p);
```
- const Value Parameter that is Pointer
```
void function(const double* p);
```
- Reference Parameter that is Pointer
```
void function(double*& p);
```

THE BAG CLASS WITH A DYNAMIC ARRAY

Dynamic Data Structure

- We can use a pointer to define dynamic data structure
- Dynamic data structures
  - Data structures whose size is determined when a program is actually running rather than at compilation time
  - Static data structures have their size determined when a program is compiled
  - A class can be a dynamic data structure, i.e., it may use dynamic memory

Pointer Member Variables

- The original bag class has a member variable that is a static array containing the bag's items
- Our dynamic bag has a member variable that is a pointer to a dynamic array

The Static Bag:

```
class bag
{
    ...
private:
    value_type data[CAPACITY];
    size_type used;
};
```

The Dynamic Bag:

```
class bag
{
    ...
private:
    value_type *data;
    size_type used;
};
```

Pointer Member Variables (Cont.)

- The constructor for the dynamic bag will allocate a dynamic array
- As a program runs, a new, larger dynamic array can be allocated

```
class bag
{
  public:
    ...
  private:
    value_type *data;
    size_type used;
    size_type capacity;
};
```

Points to a partially filled dynamic array that stores the actual items of the bag

Stores the number of items in the bag

Stores the total size of the dynamic array

Member Functions Allocate Dynamic Memory As Needed

- The class's member functions allocate dynamic memory as needed
- The constructor of the dynamic bag allocates the dynamic array that the member variable *data* points to

- Question: How big should this array be?
  - Our plan is to have the constructor allocate a dynamic array whose initial size is determined by a parameter to the constructor
  - Whenever items are inserted into a bag (through the insert member function or the += operator), the bag's capacity may be increased

Member Functions Allocate Dynamic Memory As Needed (cont.)

- Start with a small initial capacity and insert items one after another
  - The insert function will take care of increasing the capacity as needed
  - Yes, this approach works correctly
- However, if there are many items, many of the activations of insert would need => **inefficient**
  - Each time the capacity is increased, new memory is allocated, the items are copied into the new memory, and the old memory is released
- To avoid this repeated allocation of memory, a programmer can request a large initial capacity

Member Functions Allocate Dynamic Memory As Needed (cont.)

- The new bag's constructor:

```
bag(size_type initial_capacity = DEFAULT_CAPACITY);
// Postcondition: The bag is empty with a capacity given by the
// parameter.
// The insert function will work efficiently (without allocating
// new memory) until this capacity is reached.
```

- When the bag is declared, the programmer can specify a capacity of 1000:

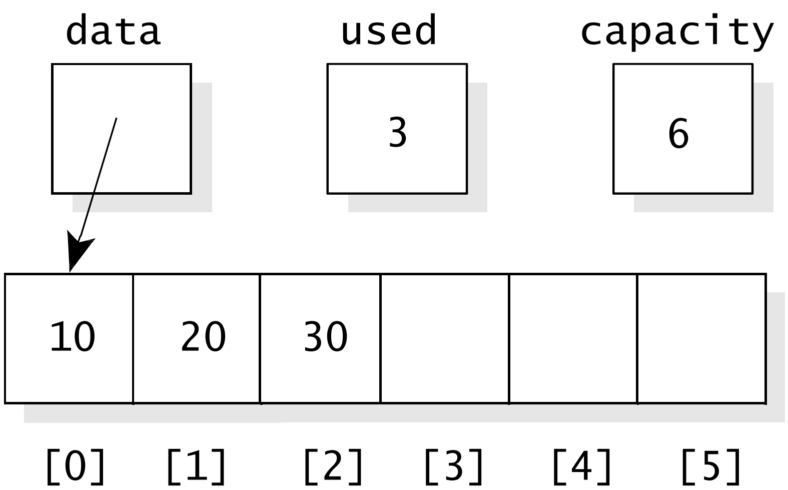
```
bag kilosack(1000);
```

- After the initial capacity is reached, the insert function continues to work correctly, but it might be slowed down by memory allocations

Member Functions Allocate Dynamic Memory As Needed (Cont.)

```
bag sixpack(6);

sixpack.insert(10);
sixpack.insert(20);
sixpack.insert(30);
```

After these declarations, the bag's private member variables look like this:



Member Functions Allocate Dynamic Memory As Needed (Cont.)

- While the bag is in use, a programmer can make an explicit adjustment to the bag's capacity via a member function called `reserve`:

```
void reserve(size_type new_capacity);
// Postcondition: The bag's current capacity is changed to the
// new_capacity (but not less than the number of items already in
// the bag).
// The insert function will work efficiently (without allocating
// new memory) until the new capacity is reached.
```

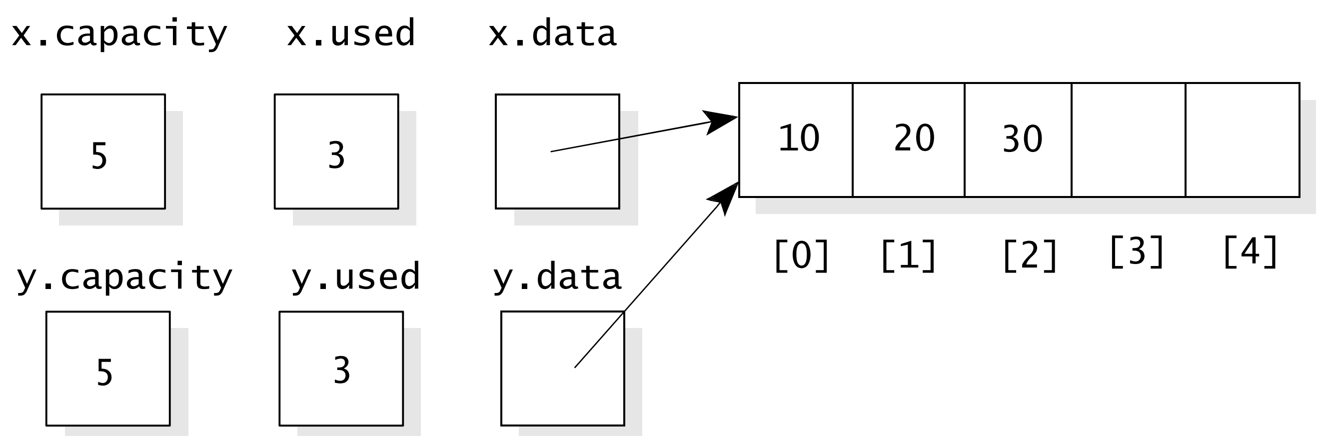
- The constructor, reserve, insert, += operator and + operator member functions can allocate new dynamic memory
  - include documentation to indicate which member functions allocate dynamic memory

Value Semantics

- With all our other classes, it was sufficient to use the automatic assignment operator and the automatic copy constructor
- **Automatic assignment**: y = x
  - copy all the member variables from x to y
- The automatic assignment operator fails for the dynamic bag (or for any other class that uses dynamic memory)

Value Semantics (cont.)

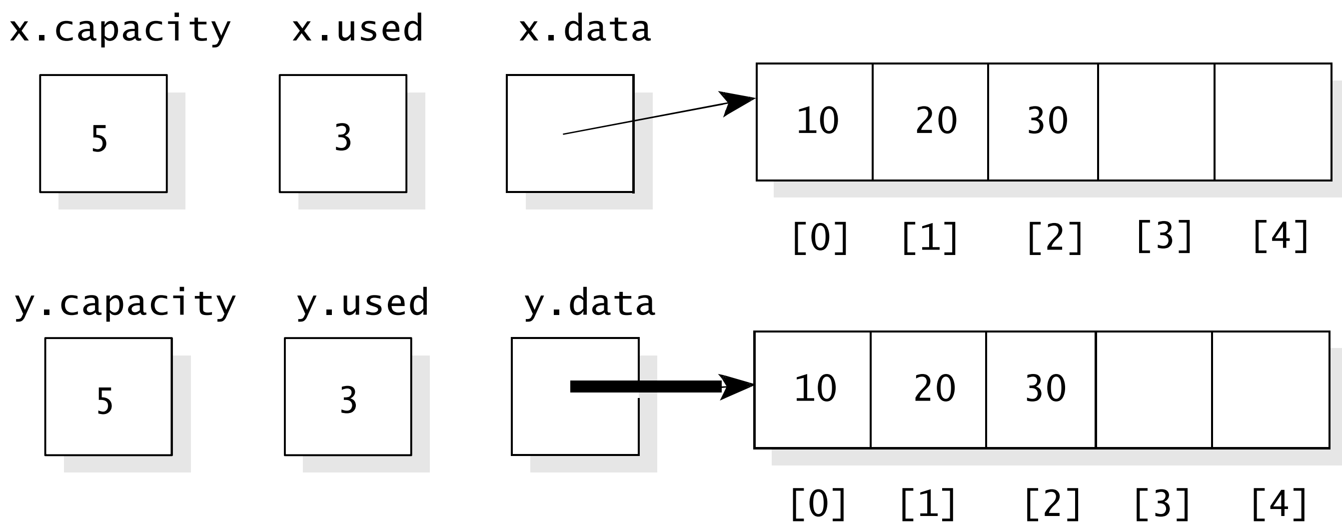
- Suppose a bag called x has an initial capacity of 5, containing the integers 10, 20, and 30
- x is assigned to y through y = x



What is the problem?

Value Semantics (cont.)

- y need to have its own dynamic array, completely separate from x's dynamic array



- Provide our own assignment operator rather than relying on the automatic assignment operator
- Do this by overloading the assignment operator for the bag class

Value Semantics (cont.)

```
void bag::operator = (const bag& source);
// Postcondition: The bag that activated this function has the
// same items and capacity as source.
```

- When you overload the assignment operator, C++ requires it to be a member function
- In an assignment statement y = x, the bag y is activating the function, and the bag x is the argument for the parameter named source

Value Semantics (cont.)

- **Copy constructor** is activated when a new object is initialized as a copy of an existing object
- bag y(x);
  - y is initialized using the automatic copy constructor, which merely copies the member variables from x to y
  - To avoid the simple copying of member variables, a copy constructor with the prototype is needed

```
bag::bag(const bag& source);
// Postcondition: The bag that is being constructed has been
// initialized with the same items and capacity as source.
```

Note: The parameter of the copy constructor is usually a const reference parameter (C++ also permits an ordinary reference parameter, but does not allow a value parameter)

The Destructor

- The destructor of a class is a member function
  `~bag( );`
- Automatically activated when an object becomes inaccessible
  - The primary purpose of the destructor is to return an object’s dynamic memory to the heap when the object is no longer in use
- Programmers who use a class should not need to know about the destructor
- The activation is usually automatic whenever an object becomes inaccessible
  - Programs rarely activate the destructor explicitly

The Destructor (cont.)

- Several common situations cause automatic destructor activation:
- When a local variable is an object with a destructor, the destructor is automatically activated when the function returns

```
void example1( )
{
    bag sample1;
    ...
}
```

- When the function example1 returns, the destructor sample1.~bag( ) is automatically activated

The Destructor (cont.)

- Suppose a function has a value parameter that is an object

```
void example2(bag sample2)
// Does some calculation using a bag
```

- When the function example2 returns, the destructor sample2.~bag( ) is automatically activated

Note: If sample2 was a reference parameter, then the destructor would not be activated because a reference parameter is actually an object in the calling program, and that object is still accessible

The Destructor (cont.)

- Suppose that a dynamic variable is an object

```
bag *b_ptr;
b_ptr = new bag;
...
delete b_ptr;
```

- When delete b_ptr is executed, the destructor for *b_ptr is automatically activated
- The destructor ensures that the dynamic array used by *b_ptr is released

Header File for the Bag Class with a Dynamic Array