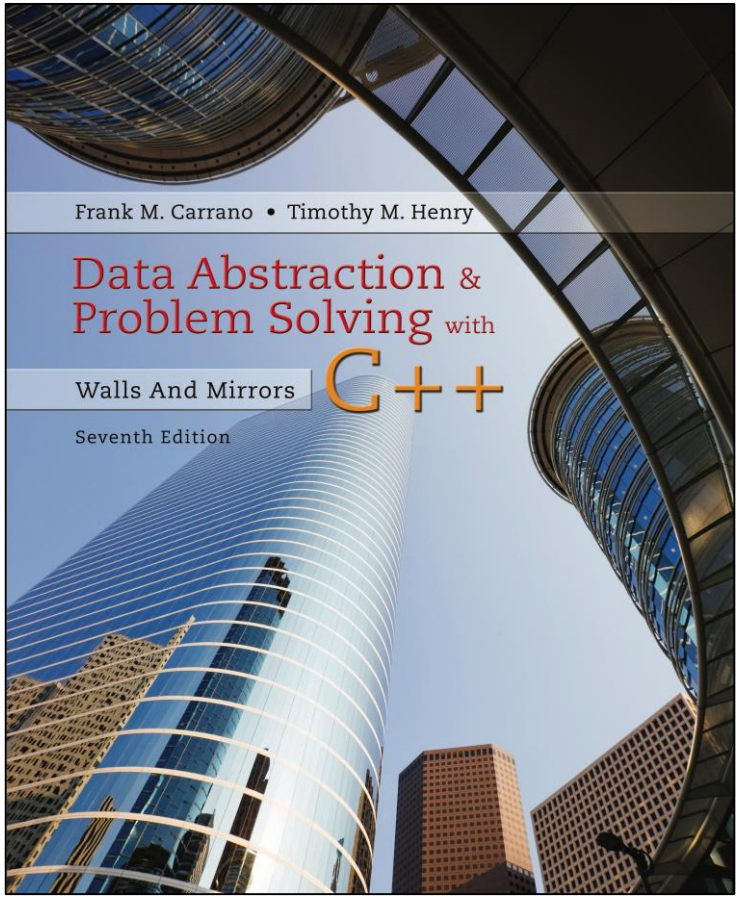
# Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition




## C++ Interlude 2

Pointers, Polymorphism, and Memory Allocation

# Memory Allocation for Variables and Early Binding of Methods (1 of 3)

- Declare variable *x* to have data type **int**
  - C++ compiler allocates memory cell to hold an integer
  - Use the identifier *x* to refer to this cell
- A function's locally declared variables
  - Placed into an activation record with parameters and bookkeeping data
  - Activation record placed on run-time stack
  - Activation record destroyed when function finished

# Memory Allocation for Variables and Early Binding of Methods (2 of 3)

- Storage for data members of an object
  - Also placed into an activation record.
  - Data fields placed on the run-time stack just as primitive data types are.
- This is early binding , made during compilation
  - Cannot be altered during execution

# Memory Allocation for Variables and Early Binding of Methods (3 of 3)

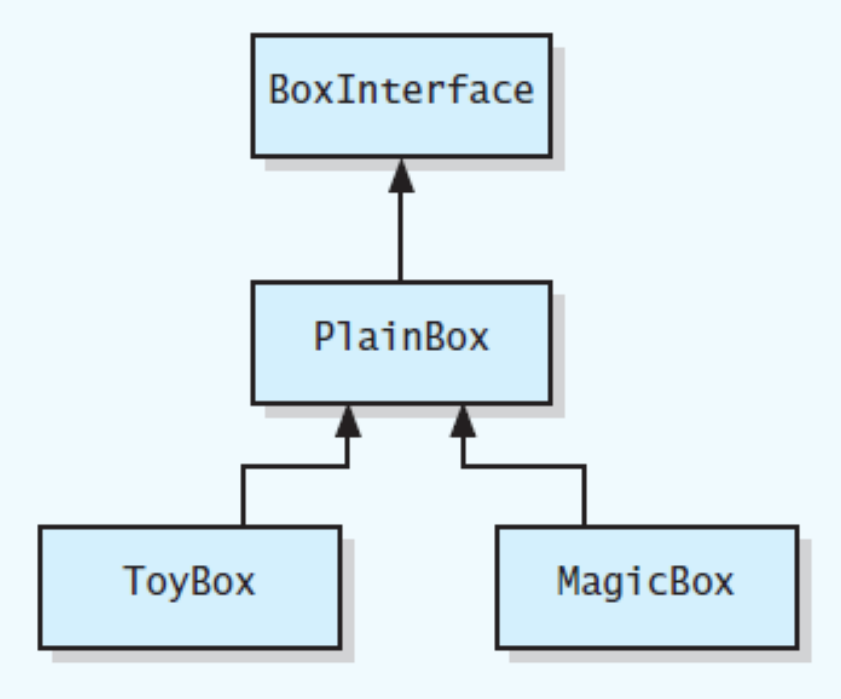
- Automatic memory management and early binding sometimes insufficient
  - Need to take advantage of polymorphism.
  - Must access an object outside of the function or method that creates it.

# Problem to Solve (1 of 5)

- Need to write a function – takes two arguments:
  - An object of any of the three types of boxes (from Interlude 1)
  - An item of type **string**
- Function should place item in box by invoking box's **setItem** method

# Problem to Solve (2 of 5)

**Figure C2-1** UML class diagram for a family of classes



# Problem to Solve (3 of 5)

- You may think this function would suffice

```
void placeInBox(PlainBox<string>& theBox, string theItem)
{
   theBox.setItem(theItem);
}  // end placeInBox
```

# Problem to Solve (4 of 5)

- Used in this context

```
std::string specialItem = "Riches beyond compare!";
std::string hammerItem = "Hammer";

PlainBox<std::string> myPlainBox;
placeInBox(myPlainBox, hammerItem);              // Stores hammerItem
placeInBox(myPlainBox, specialItem);             // Stores specialItem
std::cout << myPlainBox.getItem() << std::endl;  // Displays specialItem

MagicBox<std::string> myMagicBox;
placeInBox(myMagicBox, hammerItem);              // Stores hammerItem
placeInBox(myMagicBox, specialItem);             // Stores specialItem
std::cout << myMagicBox.getItem() << std::endl;  // Displays specialItem
```

- Code compiles, but does not perform as you would expect due to

# Problem to Solve (5 of 5)

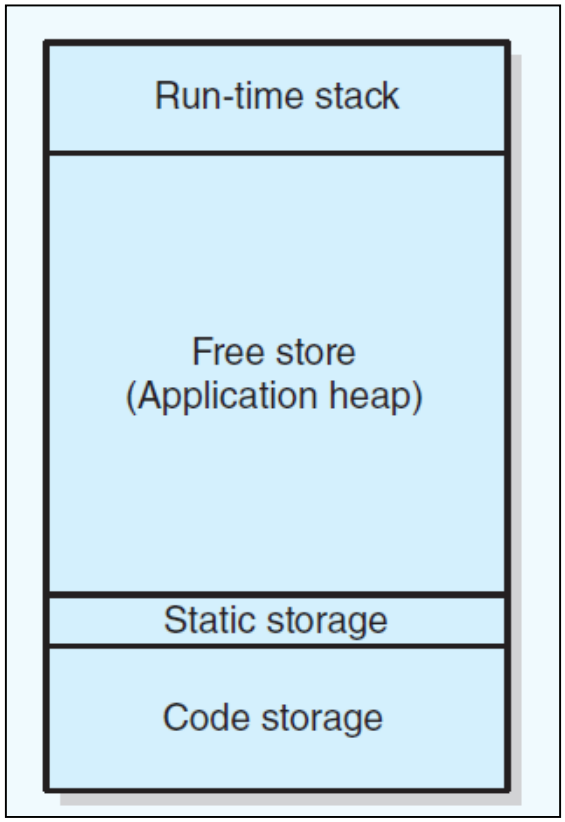
- Version of **setItem** called is determined when the program is compiled.

```
void placeInBox(PlainBox<string>& theBox, string theItem)
{
   theBox.setItem(theItem);
}  // end placeInBox
```

- Need a way to communicate to compiler
  - Code to execute should not be determined until program is running.
  - Called late binding

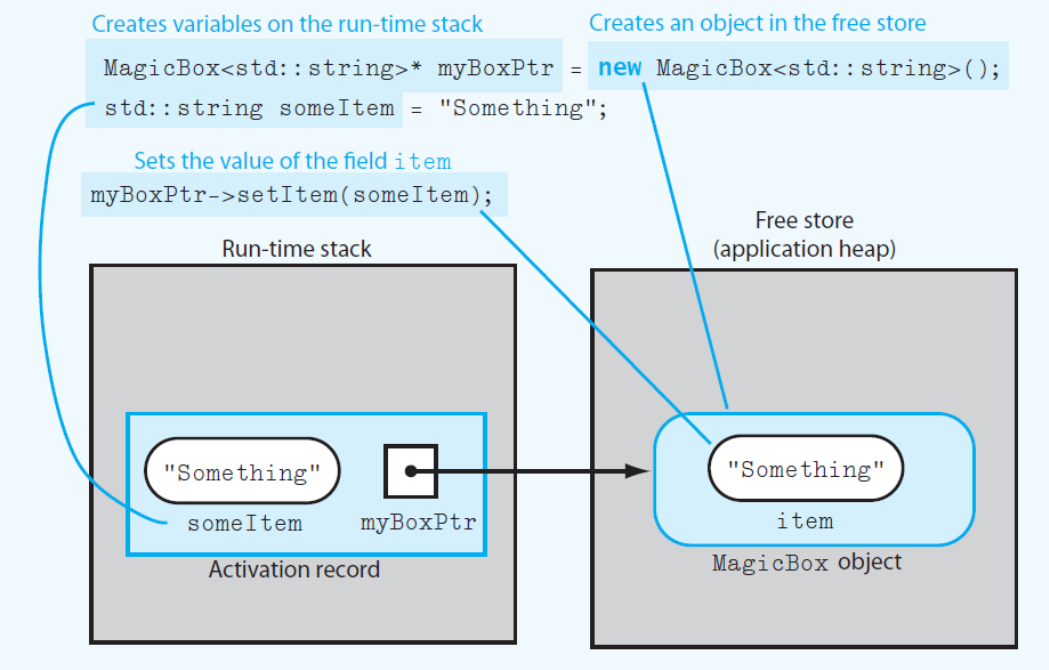
# Pointers and Program's Free Store (1 of 4)

**Figure C2-2** Sample program memory layout



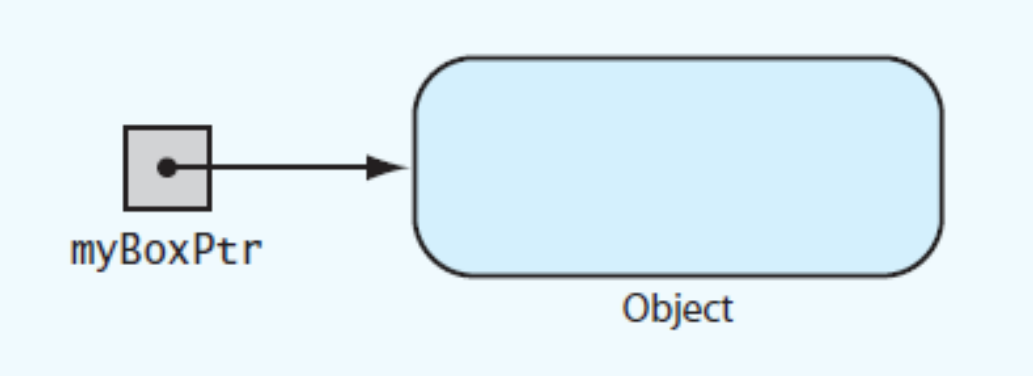
# Pointers and Program's Free Store (2 of 4)

**Figure C2-3** Run-time stack and free store after **myboxPtr** points to a **MagicBox** object and its data member item is set



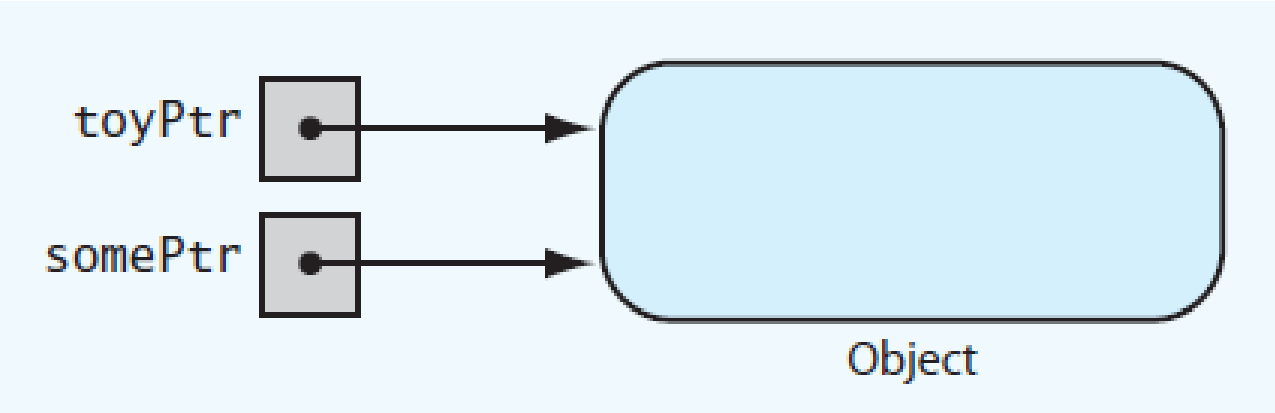
# Pointers and Program's Free Store (3 of 4)

**Figure C2-4 myBoxPtr** and the object to which it points



# Pointers and Program's Free Store (4 of 4)

**Figure C2-5** Two pointer variables that point to the same object



# Deallocating Memory

- When memory to which pointer variable points is no longer needed
  - Deallocate it by using **delete** operator.
- Then set pointer variable to **nullptr**
- Otherwise dangling pointer exists
  - It would still contain address of object that was deallocated.
  - Can be source of serious errors.

# Avoiding Memory Leaks (1 of 4)

- Memory leaks occur when
  - An object has been created in the free store, but
  - Program no longer has a way to access

**Listing C2-1** Poorly written function that allocates memory in the free store

```
void myLeakyFunction(const double& someItem)
{
   ToyBox<double>* someBoxPtr = new ToyBox<double>();
   someBoxPtr->setItem(someItem);
}  // end myLeakyFunction
```

# Avoiding Memory Leaks (2 of 4)

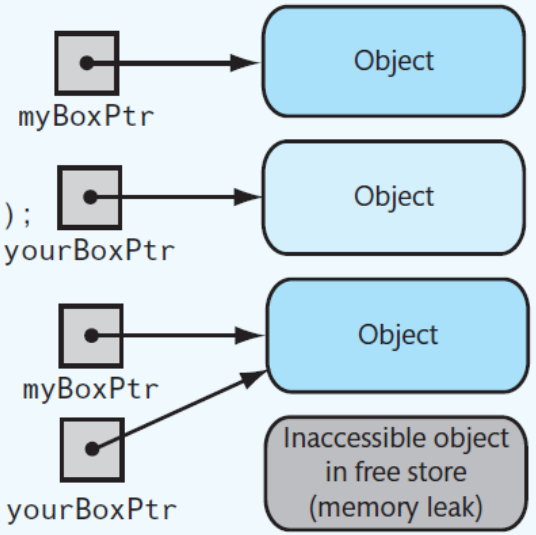
**Figure C2-6** An assignment that causes an inaccessibl object.

- To prevent memory leak, do not use a function to return a pointer to a newly created object

```
// Creating first object
MagicBox<std::string>* myBoxPtr = new MagicBox<std::string>();

// Creating second object
MagicBox<std::string>* yourBoxPtr = new MagicBox<std::string>();

// Assignment causes an inaccessible object
yourBoxPtr = myBoxPtr;
```



# Avoiding Memory Leaks (3 of 4)

**Listing C2-2** Header file for the class **GoodMemory**

```
1  /** @file GoodMemory.h */
2  #ifndef GOOD_MEMORY_
3  #define GOOD_MEMORY_
4  #include "ToyBox.h"
5
6  class GoodMemory
7  {
8  private:
9     ToyBox<double>* someBoxPtr;
10 public:
11    GoodMemory();           // Default constructor
12    virtual ~GoodMemory(); // Destructor
13    void fixedLeak(const double& someItem);
14 }; // end GoodMemory
15 #endif
```

# Avoiding Memory Leaks (4 of 4)

**Listing C2-3** Implementation file for the class **GoodMemory**

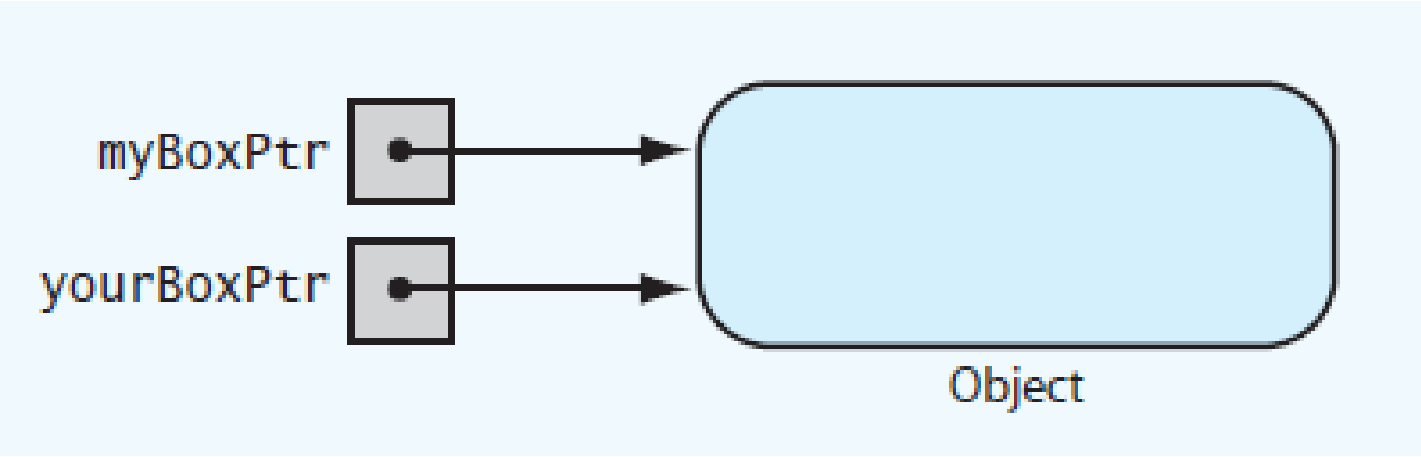
```
1  /** @file GoodMemory.cpp */
2  #include "GoodMemory.h"
3
4  GoodMemory::GoodMemory() : someBoxPtr(nullptr)
5  {
6  }  // end default constructor
7
8  GoodMemory::~GoodMemory()
9  {
10    delete someBoxPtr;
11 }  // end destructor
12
13 void GoodMemory::unleakyMethod(const double& someItem)
14 {
15    someBoxPtr = new ToyBox<double>();
16    someBoxPtr->setItem(someItem);
17 }  // end unleakyMethod
```

# Avoiding Dangling Pointers (1 of 3)

- Situations that can cause dangling pointer
  - if you do not set a pointer variable to **nullptr** after using **delete**
  - If you declare a pointer variable but do not assign it a value

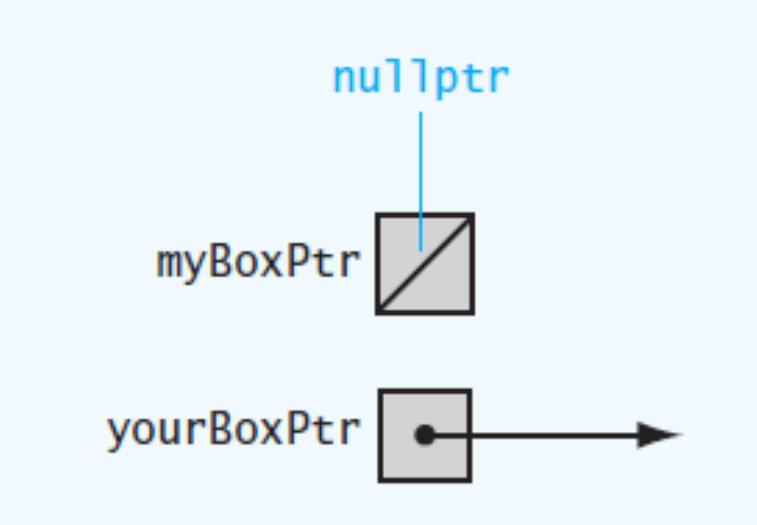
# Avoiding Dangling Pointers (2 of 3)

**Figure C2-7** Two pointers referencing (pointing to) the same object



# Avoiding Dangling Pointers (3 of 3)

**Figure C2-8** Example of a dangling pointer



# Virtual Methods and Polymorphism (1 of 4)

- Allow compiler to perform the late binding necessary for polymorphism
  - Declare methods in base class as **virtual**.
- **Listing C2-4** Revised header file for the class **PlainBox**

```
1  /** @file PlainBox.h */
2  #ifndef PLAIN_BOX_
3  #define PLAIN_BOX_
4
5  template<class ItemType> // Indicates this is a template
6
7  // Declaration for the class PlainBox
8  class PlainBox
9  {
10 private:
11    // Data field
12    ItemType item;
```

# Virtual Methods and Polymorphism (2 of 4)

**Listing C2-4** Revised header file for the class **PlainBox**

```
14  public:
15     // Default constructor
16     PlainBox();
17
18     // Parameterized constructor
19     PlainBox(const ItemType& theItem);
20
21     // Mutator method that can change the value of the data field
22     virtual void setItem(const ItemType& theItem);
23
24     // Accessor method to get the value of the data field
25     virtual ItemType getItem() const;
26  }; // end PlainBox
27
28  #include "PlainBox.cpp" // Include the implementation file
29  #endif
```

# Virtual Methods and Polymorphism (3 of 4)

**Listing C2-4** Revised header file for the class **PlainBox**

```
   // Data field
   ItemType item;

public:
   // Default constructor
   PlainBox();

   // Parameterized constructor
   PlainBox(const ItemType& theItem);

   // Mutator method that can change the value of the data field
   virtual void setItem(const ItemType& theItem);

   // Accessor method to get the value of the data field
   virtual ItemType getItem() const;
}; // end PlainBox

#include "PlainBox.cpp" // Include the implementation file
#endif
```

# Virtual Methods and Polymorphism (4 of 4)

To fully implement late binding, create variables in free store and use pointers to reference them

```
std::string specialItem = "Riches beyond compare!";
std::string hammerItem = "Hammer";

PlainBox<std::string>* myPlainBoxPtr = new PlainBox<std::string>();
placeInBox(myPlainBoxPtr, hammerItem);              // Stores hammerItem
placeInBox(myPlainBoxPtr, specialItem);             // Stores specialItem
std::cout << myPlainBoxPtr->getItem() << std::endl; // Displays specialItem

MagicBox<std::string>* myMagicBoxPtr = new MagicBox<std::string>();
placeInBox(myMagicBoxPtr, hammerItem);              // Stores hammerItem
placeInBox(myMagicBoxPtr, specialItem);             // Ignores specialItem
std::cout << myMagicBoxPtr->getItem() << std::endl; // Displays hammerItem
```

# Dynamic Allocation of Arrays (1 of 2)

- An ordinary C++ array is statically allocated

```
const int MAX_SIZE = 50;
double myArray[MAX_SIZE];
```

- Can use **new** operator to allocate an array dynamically

```
int arraySize = 50;
double* anArray = new double[arraySize];
```

# Dynamic Allocation of Arrays (2 of 2)

- **delete** returns a dynamically allocated array to system for reuse

```
delete [ ] anArray;
```

- Increase size of dynamically allocated array

```
double* oldArray = anArray;                    // Copy pointer to array
anArray = new double[2 * arraySize];           // Double array size

for (int index = 0; index < arraySize; index++)    // Copy old array
   anArray[index] = oldArray[index];

delete [ ] oldArray;                           // Deallocate old array
```

# A Resizable Array-Based Bag (1 of 2)

Use a resizable array to implement ADT bag so that bag never becomes full

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
   bool hasRoomToAdd = (itemCount < maxItems);
   if (!hasRoomToAdd)
   {
      ItemType* oldArray = items;
      items = new ItemType[2 * maxItems];
      for (int index = 0; index < maxItems; index++)
         items[index] = oldArray[index];
```

# A Resizable Array-Based Bag (2 of 2)

Use a resizable array to implement ADT bag so that bag never becomes full

```
            items[index] = oldArray[index];
        delete [ ] oldArray;
        maxItems = 2 * maxItems;
    }  // end if

    // We can always add the item
    items[itemCount] = newEntry;
    itemCount++;
    return true;
}  // end ResizableArrayBag add
```

# Copyright