# Data Structures Using C++ 2E

*Chapter 3*
*Pointers and Array-Based Lists*

# 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
- Learn about pointer arithmetic

# Objectives (cont'd.)

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

# The Pointer Data Type and Pointer Variables

- Pointer data types
  - Values are computer memory addresses
  - No associated name
  - Domain consists of addresses (memory locations)
- Pointer variable
  - A variable whose content is an address (memory address)

# The Pointer Data Type and Pointer Variables (cont'd.)

- Declaring pointer variables
  - Specify data type of value stored in the memory location that pointer variable points to
  - General syntax

    `dataType *identifier;`
  - Asterisk symbol (`*`)
    - Between data type and variable name
    - Can appear anywhere between the two
    - Preference: attach `*` to variable name
  - Examples:

    `int *p; char *ch;`

    `int *a, b;` // only `a` is pointer, `b` is not

# The Pointer Data Type and Pointer Variables (cont'd.)

- Address of operator (`&`)
  - Unary operator
  - Returns address of its operand

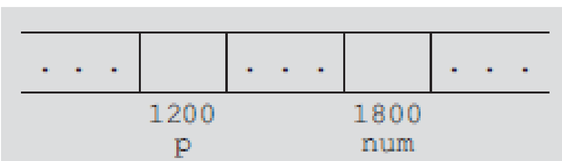
```
int *a, b; a = &b;
```

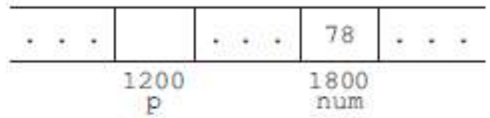
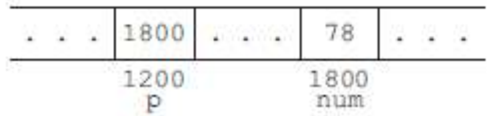
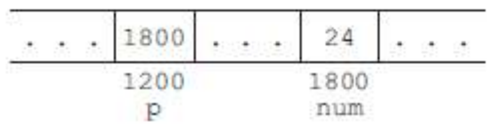
- Dereferencing operator (`*`)
  - Unary operator
    - Different from binary multiplication operator
  - Also known as indirection operator
  - Refers to object where the pointer points

```
int *a, b; a = &b; *a = 5;
```

# The Pointer Data Type and Pointer Variables (cont'd.)

1. `num = 78;`
2. `p = #`
3. `*p = 24;`




| After statement | Values of the variables | Explanation |
|---|---|---|
| 1 |   | The statement `num = 78;` stores 78 into num. |
| 2 |   | The statement `p = #` stores the address of num, which is 1800, into p. |
| 3 |   | The statement `*p = 24;` stores 24 into the memory location to which p points. Because the value of p is 1800, statement 3 stores 24 into memory location 1800. Note that the value of num is also changed. |

Let us summarize the preceding discussion.

1. A declaration such as **int** `*p`; allocates memory for p only, not for `*p`. Later, you learn how to allocate memory for `*p`.
2. The content of p points only to a memory location of type **int**.
3. &p, p, and `*p` all have different meanings.
4. &p means the address of p—that is, 1200 (as shown in Figure 3-1).
5. p means the content of p, which is 1800, after the statement `p = #` executes.
6. `*p` means the content of the memory location to which p points. Note that the value of `*p` is 78 after the statement `p = #` executes; the value of `*p` is 24 after the statement `*p = 24;` executes.

# The Pointer Data Type and Pointer Variables (cont'd.)

- Pointers and classes
  - Dot operator (`.`)
    - Higher precedence than dereferencing operator (`*`)
  - Member access operator arrow ( `->`)
    - Simplifies access of `class` or `struct` components via a pointer
    - Consists of two consecutive symbols: hyphen and "greater than" symbol
  - Syntax

```
pointerVariableName -> classMemberName
```

```
(*pointerVariableName).classMemberName
```

# The Pointer Data Type and Pointer Variables (cont'd.)

```
string *str;
str = new string;
*str = "Hello World";
// the meaning of
// (*str).length()
// *str.length()
// str->length()
cout << "length = " << (*str).length();
cout << "length = " << str->length();
```

Compilation error: left of .length must have class/struct/union

# The Pointer Data Type and Pointer Variables (cont'd.)

- Initializing pointer variables
  - No automatic variable initialization in C++
  - Pointer variables must be initialized
    - If not initialized, they do not point to anything
  - Initialized using
    - Constant value 0 (null pointer)
    - Named constant `NULL`
  - Number 0
    - Only number directly assignable to a pointer variable

```
int *p;
p = 0;  p = NULL;
```

# The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic variables
  - Variables created during program execution
    - Real power of pointers
  - Two operators
    - `new`: creates dynamic variables
    - `delete`: destroys dynamic variables
    - Reserved words
    - Use the matching one: `new`/`delete`, `malloc`/`free`

# The Pointer Data Type and Pointer Variables (cont'd.)

- Operator `new`
  - Allocates single variable, or array of variables
  - Syntax

```
new dataType;
new dataType[intExp];
```

  - Allocates memory (variable) of designated type
    - Returns pointer to the memory (allocated memory address)
    - Allocated memory: uninitialized

# The Pointer Data Type and Pointer Variables (cont'd.)

- Operator `delete`
  - Destroys dynamic variables
  - Syntax

```
delete ptrVariable;
delete [] ptrArrayVariable;
```

  - w/o `delete` ➔ memory leak (cannot be reallocated)
  - Dangling pointers
    - Pointer variables containing addresses of deallocated memory spaces
    - Avoid by setting deleted pointers to `NULL` after `delete`

# Example

```
#include <iostream>
using namespace std;

void main()
{
    cout << "hello world\n";
    int *a, b;

    a = &b;
    cout << "a = " << a << endl;
    cout << "&a = " << &a << endl;

    a = new int;
    cout << "a = " << a << endl;
    delete a;

    a = new int[10];
    cout << "a = " << a << endl;
    delete [] a;
}
```

# Example 2

- Code snippet

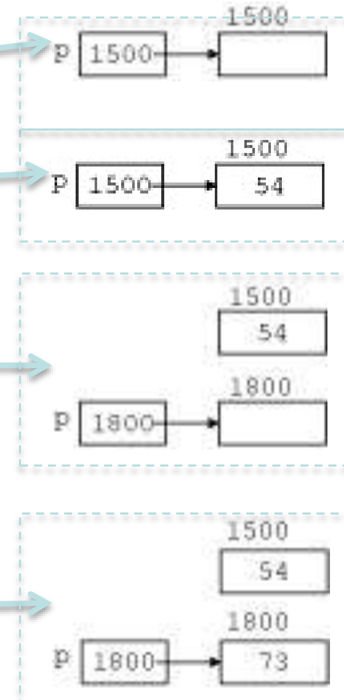
```
p = new int;   // Line 1

*p = 54;       // Line 2


p = new int;   // Line 3



*p = 73;       // Line 4
```




- What is the problem with the above 4 lines?
  - How to avoid it?
- Example 3.3, Chapter 3

# The Pointer Data Type and Pointer Variables (cont'd.)

- Operations on pointer variables
  - Operations allowed
    - Assignment, relational operations; some limited arithmetic operations
    - Assign value of one pointer variable to another pointer variable of the same type
    - Compare two pointer variables for equality, e.g.,

      `p == q` or `p != q`
    - Add and subtract integer values from pointer variable, e.g.,

      `p++` or `p = p + 2`

      If `int` is 4 bytes, increments `p`'s value by 8
  - Danger
    - Accidentally accessing other variables' memory locations and changing content without warning

# The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic arrays
  - Static array limitation
    - Fixed size
    - Not possible for same array to process different data sets of the same type
  - Solution
    - Declare array large enough to process a variety of data sets
    - Problem: potential memory waste
  - Dynamic array solution
    - Prompt for array size during program execution

# The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic arrays (cont'd.)
  - Dynamic array
    - An array created during program execution
  - Dynamic array creation
    - Use `new` operator
  - Example

```
p = new int[10];
*p = 25; p++; *p = 35;
// equivalent to p[0] = 25; p[1] = 35;
```

# The Pointer Data Type and Pointer Variables (cont'd.)

- Array name: a constant pointer
  - Array name value: constant

    ```
    int list[5];
    ```

  - Increment, decrement operations cannot be applied

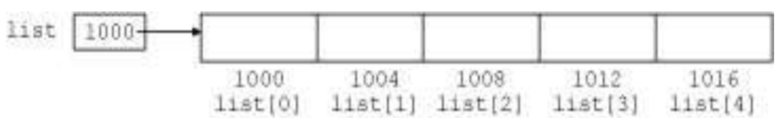



**FIGURE 3-14** `list` and array `list`

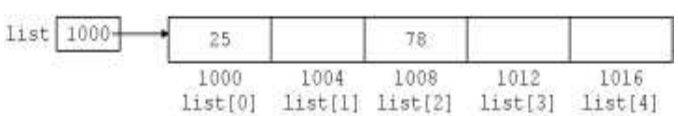



**FIGURE 3-15** Array `list` after the execution of the statements `list[0] = 25;` and `list[2] = 78;`

# Examples

- Is there anything wrong?

```
int x;
int *p;
int *q;
p = new int[10];
q = p;
* p = 4;
for( int j = 0; j < 10; j++)
{
   x = *p;
   p++;
   *p = x + j;
}
for (int k = 0; k < 10; k++)
{
   cout << *q << " ";
   q++;
}
```

```
int x;
int p[10];
int *q;
q = p;
* p = 4;
for( int j = 0; j < 9; j++)
{
   x = *p;
   p++;
   *p = x + j;
}
for (int k = 0; k < 10; k++)
{
   cout << *q << " ";
   q++;
}
```

```
int *p;
int *q;

p = new int[5];
*p = 2;

for( int j = 1; j < 5; j++)
   p[j]=p[j-1] + j;
q=p;
delete [] p;

for (int j=0; j<5; j++)
   cout << q[j] << " ";
cout << endl;
```

# The Pointer Data Type and Pointer Variables (cont'd.)

- Functions and pointers
  - Pointer variable passed as parameter to a function
    - By value or by reference
  - By value: declaring a pointer as a value parameter in a function heading
    - Same mechanism used to declare a variable
  - By reference: making a formal parameter be a reference parameter
    - Use `&` when declaring the formal parameter in the function heading

# The Pointer Data Type and Pointer Variables (cont'd.)

- Functions and pointers (cont'd.)
  - Formal parameter as reference parameter: `&`
    - Between data type name and identifier name
  - Formal parameter as pointer: `*`
    - Between data type name and identifier name
  - Reference parameter as pointer: `* &`

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

`p`: pointer, reference parameter
`q`: pointer, value parameter

# The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic two-dimensional arrays
  - Creation
    - 4 x 6: 4 rows and 6 columns

```
int *board[4];
for (int row = 0; row < 4; row++)
board[row] = new int[6];
```

# The Pointer Data Type and Pointer Variables (cont'd.)

- Dynamic two-dimensional arrays (cont'd.)
  - Declare `board` to be a pointer to a pointer

```
int **board;
```

  - Declare `board` to be an array of 10 rows and 15 columns
    - To access `board` components, use array subscripting notation

```
board = new int* [10];
for (int row = 0; row < 10; row++)
   board[row] = new int[15];
```

# The Pointer Data Type and Pointer Variables (cont'd.)

- Shallow vs. deep copy
  - Shallow copy: copying the pointer values only
    - Two or more pointers of same type
    - Points to same memory
    - Points to same data
      - Dangling pointer if memory is freed via the other pointer

# The Pointer Data Type and Pointer Variables (cont'd.)

- Shallow copy

```
int *first;

int *second;
```

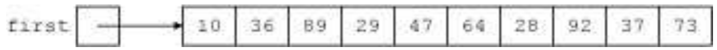



**FIGURE 3-16** Pointer `first` and its array


**FIGURE 3-17** `first` and `second` after the statement `second = first;` executes

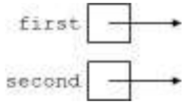



**FIGURE 3-18** `first` and `second` after the statement `delete [] second;` executes

# The Pointer Data Type and Pointer Variables (cont'd.)

- Deep copy: copying both pointer values and the data they point to
  - Two or more pointers have their own data

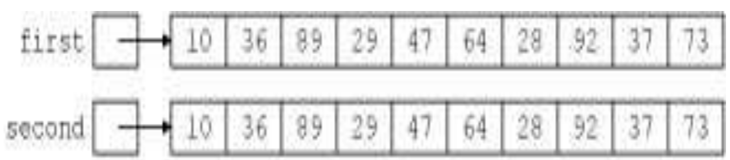



**FIGURE 3-19** `first` and `second` both pointing to their own data

```
for (int i = 0; i < size; i++)
  second[i] = first[i];
```

  - Used in overloading assignment operator and overriding copy constructor

# Classes and Pointers: Some Peculiarities

- Class can have pointer member variables
  - Peculiarities of such classes exist

```
class pointerDataClass
{
public:
      .
      .
      .

private:
    int x;
    int lenP;
    int *p;
};
```

What if p points to a dynamic array? How to deallocate?

Also consider the following statements. (See Figure 3-20.)

```
pointerDataClass objectOne;
pointerDataClass objectTwo;
```

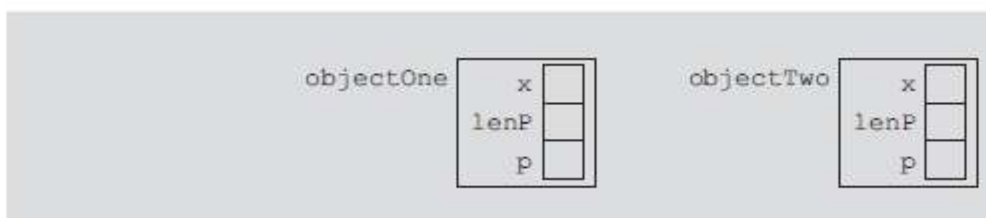
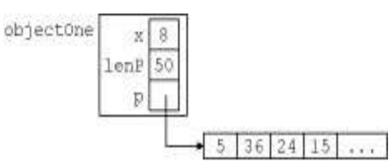



**FIGURE 3-20** Objects `objectOne` and `objectTwo`

# Classes and Pointers: Some Peculiarities (cont'd.)

- Destructor
  - Could be used to prevent an array from staying marked as allocated
    - Even though it cannot be accessed
  - If a `class` has a destructor
    - Destructor automatically executes whenever a `class` object goes out of scope
    - Put code in destructor to deallocate memory




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

**FIGURE 3-21** Object `objectOne` and its data

# Classes and Pointers: Some Peculiarities (cont'd.)

- Assignment operator
  - *Built-in* assignment operators for classes with pointer member variables may lead to shallow copying of data

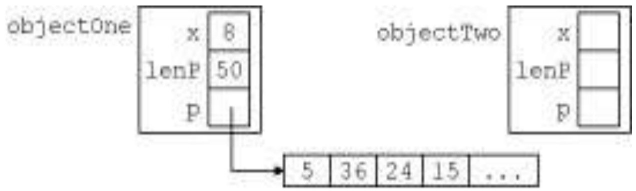



**FIGURE 3-22** Objects `objectOne` and `objectTwo`

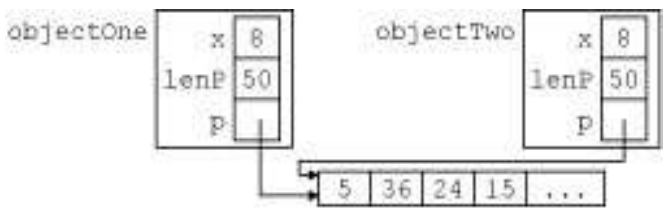



**FIGURE 3-23** Objects `objectOne` and `objectTwo` after the statement `objectTwo = objectOne;` executes

# Classes and Pointers: Some Peculiarities (cont'd.)

- Assignment operator (cont'd.)
  - **Overloading the assignment operator**
    - Deep copy
    - Avoids shallow copying of data for classes with a pointer member variable

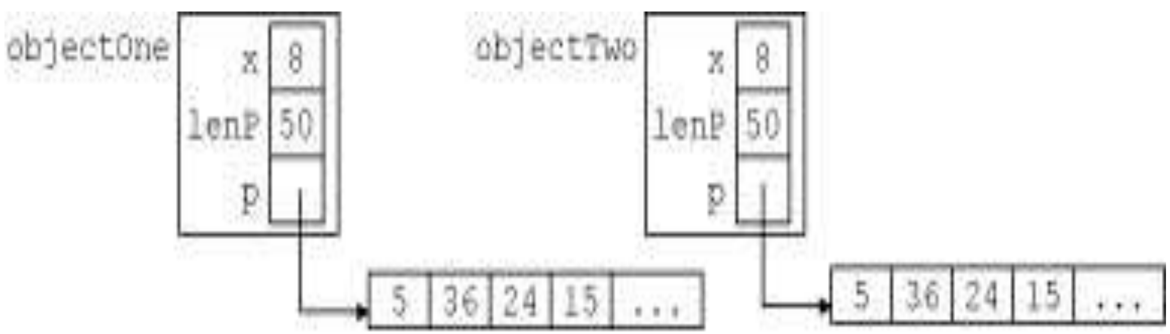



**FIGURE 3-24** Objects `objectOne` and `objectTwo`

# Classes and Pointers: Some Peculiarities (cont'd.)

- Copy constructor
  - When declaring the class object
    - Can initialize a class object by using the value of an existing object of the same type
  - Default memberwise initialization
    - May result from copy constructor provided by compiler
    - May lead to shallow copying of data
    - Solution: **overriding copy constructor**
  - Syntax to include copy constructor in the definition of a class

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

# Classes and Pointers: Some Peculiarities (cont'd.)

Not assignment operator executed

- Copy constructor automatically executes
  - when an object is declared and initialized using the value of another object of the same type

    `objType a = b;  //` equivalent to `objType a(b);`
  - when, as a parameter, an object is passed by value
  - when the return value of a function is an object
- For classes with pointer member variables
  - Include the destructor in the class
  - Overload the assignment operator for the class
  - Include the copy constructor

# Inheritance, Pointers, and Virtual Functions

- Class object can be passed either by value or by reference
- C++ allows passing of an object of a derived class to a formal parameter of the base class type
- Formal parameter: reference parameter or a pointer
  - Compile-time binding: compiler generates code to call a specific function
  - Run-time binding: compiler does not generate code to call a specific function
  - Virtual functions: enforce run-time binding of functions

```cpp
class baseClass
{
public:
    void print();
    baseClass(int u = 0);

private:
    int x;
};

class derivedClass: public baseClass
{
public:
    void print();
    derivedClass(int u = 0, int v = 0);

private:
    int a;
};
```

```cpp
void baseClass::print()
{
    cout << "In baseClass x = " << x << endl;
}

baseClass::baseClass(int u)
{
    x = u;
}

void derivedClass::print()
{
    cout << "In derivedClass ***: ";
    baseClass::print();
    cout << "In derivedClass a = " << a << endl;
}

derivedClass::derivedClass(int u, int v)
                : baseClass(u)
{
    a = v;
}

void callPrint(baseClass& p)
{
    p.print();
}
```

```
int main()                                   //Line 1
{                                            //Line 2
    baseClass one(5);                        //Line 3
    derivedClass two(3, 15);                 //Line 4

    one.print();                             //Line 5
    two.print();                             //Line 6

    cout << "*** Calling the function "
         << "callPrint  ***" << endl;        //Line 7

    callPrint(one);                          //Line 8
    callPrint(two);                          //Line 9

    return 0;                                //Line 10
}                                            //Line 11
```

**Sample Run:**

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint  ***
In baseClass x = 5
In baseClass x = 3
```

Compile-time binding

```
class baseClass
{
public:
    virtual void print();        //virtual function
    baseClass(int u = 0);

private:
    int x;
};
```

“virtual” in baseClass only

```
class derivedClass: public baseClass
{
public:
    void print();
    derivedClass(int u = 0, int v = 0);

private:
    int a;
};
```

**Sample Run:**

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint  ***
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
```

Run-time binding

Run-time binding also applies when a formal parameter is a pointer to a class, and a pointer of the derived class is passed as an actual parameter

```
int main()                                          //Line 5
{                                                   //Line 6
    baseClass *q;                                   //Line 7
    derivedClass *r;                                //Line 8
    q = new baseClass(5);                           //Line 9
    r = new derivedClass(3, 15);                    //Line 10

    q->print();                                     //Line 11
    r->print();                                     //Line 12

    cout << "*** Calling the function "
         << "callPrint  ***" << endl;               //Line 13

    callPrint(q);                                   //Line 14
    callPrint(r);                                   //Line 15

    return 0;                                       //Line 16
}                                                   //Line 17

void callPrint(baseClass *p)
{
    p->print();
}
```

**Sample Run:**

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint  ***
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
```

Run-time binding

# Inheritance, Pointers, and Virtual Functions (cont'd.)

- If a formal parameter of type base class is either a reference parameter or a pointer, and the function is a virtual function in the base class, we can pass a derived class object as an actual parameter and enable run-time binding
- If the formal parameter is a value parameter, then the above does not work. Why?
  - Value parameter (of base class type)→ the value of actual parameter is copied into formal parameter (of base class type)

```
int main()                                       //Line 5
{                                                //Line 6
    baseClass one(5);                            //Line 7
    derivedClass two(3, 15);                     //Line 8

    one.print();                                 //Line 9
    two.print();                                 //Line 10

    cout << "*** Calling the function "
         << "callPrint  ***" << endl;            //Line 11

    callPrint(one);                              //Line 12
    callPrint(two);                              //Line 13

    return 0;                                    //Line 14
}                                                //Line 15

void callPrint(baseClass p)  //p is a value parameter
{
    p.print();
}
```

**Sample Run:**

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint  ***
In baseClass x = 5
In baseClass x = 3
```

# Inheritance, Pointers, and Virtual Functions (cont'd.)

- Classes and virtual destructors
  - Classes with pointer member variables should have a destructor
    - Destructor automatically executed when class object goes out of scope
    - If a derived class object is passed to a formal parameter of the base class type, base class destructor executed regardless of whether derived class object passed by reference or by value
    - Derived class destructor should be executed when derived class object goes out of scope
  - Solution: `virtual` destructor

# Inheritance, Pointers, and Virtual Functions (cont'd.)

- Classes and virtual destructors (cont'd.)
  - Base class `virtual` destructor automatically makes the derived class destructor `virtual`
  - If a base class contains virtual functions
    - Make base class destructor `virtual`

# Inheritance, Pointers, and Virtual Functions (cont'd.)

```
#include <iostream>
class Base
{public:
 Base() {cout<<"Constructing Base\n";}
 ~Base(){cout<<"Destroying Base\n";}
};
class Derive: public Base
{public:
 Derive(){cout<<"Constructing Derive\n";}
 ~Derive(){cout<<"Destroying Derive\n";}
};
void main() {
   Base *basePtr = new Derive();
   delete basePtr;
}
```

Constructing Base
Constructing Derive
Destroying Base

```
#include <iostream>
class Base
{public:
 Base() {cout<<"Constructing Base\n";}
 virtual ~Base(){cout<<"Destroying
   Base\n";}
};
class Derive: public Base
{public:
 Derive(){ cout<<"Constructing Derive\n";}
 ~Derive(){ cout<<"Destroying Derive\n";}
};
void main() {
   Base *basePtr = new Derive();
   delete basePtr;
}
```

Constructing Base
Constructing Derive
Destroying Derive
Destroying Base

# Abstract Classes and Pure Virtual Functions

- Virtual functions enforce run-time binding of functions
- Inheritance
  - Allows deriving of new classes without designing them from scratch
  - Derived classes
    - Inherit existing members of base class
    - Can add their own members
    - Can redefine or override public and protected base class member functions
  - Base class can contain functions each derived class can implement

# Abstract Classes and Pure Virtual Functions (cont'd.)

- Virtual functions enforce run-time binding of functions (cont'd.)
  - Virtual functions
    - Implementation is needed
  - Pure virtual functions
    - No implementation is needed
  - Abstract class
    - Class contains one or more pure virtual functions
    - Not a complete class: cannot create objects of that class
    - Can contain instance variables, constructors, functions not pure virtual

```
class shape
{
public:
    virtual void draw() = 0;
      //Function to draw the shape. Note that this is a
      //pure virtual function.

    virtual void move(double x, double y) = 0;
      //Function to move the shape at the position (x, y).
      //Note that this is a pure virtual function.
    .
    .
    .
};
```

# Array-Based Lists

- List
  - Collection of elements of same type
- Length of a list
  - Number of elements in the list
- Many operations may be performed on a list
- Store a list in the computer's memory
  - Using an array

# Array-Based Lists (cont'd.)

- Three variables needed to maintain and process a list in an array
  - The array holding the list elements
  - A variable to store the current length of the list
  - A variable to store array max size
- Desirable to develop generic code
  - Used to implement any type of list in a program
  - Make use of templates

# Array-Based Lists (cont'd.)

- Define class implementing list as an abstract data type (ADT)

# Array-Based Lists (cont'd.)

- Definitions of functions `isEmpty`, `isFull`, `listSize` and `maxListSize`

```
template <class elemType>
bool arrayListType<elemType>::isEmpty() const
{
return (length == 0);
}

template <class elemType>
bool arrayListType<elemType>::isFull() const
{
return (length == maxSize);
}

template <class elemType>
int arrayListType<elemType>::listSize() const
{
return length;
}

template <class elemType>
int arrayListType<elemType>::maxListSize() const
{
return maxSize;
}
```

# Array-Based Lists (cont'd.)

- Template `print` (outputs the elements of the list) and template `isItemAtEqual`

```
template <class elemType>
void arrayListType<elemType>::print() const
{
   for (int i = 0; i < length; i++)
      cout << list[i] << " ";

   cout << endl;
}
template <class elemType>
bool arrayListType<elemType>::isItemAtEqual
                                  (int location, const elemType&
                                  item) const
{
   return(list[location] == item);
}
```

# Array-Based Lists (cont'd.)

- Template `insertAt`

```
template <class elemType>
void arrayListType<elemType>::insertAt
                  (int location, const elemType& insertItem)
{
   if (location < 0 || location >= maxSize)
      cerr << "The position of the item to be inserted "
           << "is out of range" << endl;
   else
        if (length >= maxSize) //list is full
             cerr << "Cannot insert in a full list" << endl;
        else
        {
             for (int i = length; i > location; i--)
                  list[i] = list[i - 1]; //move the elements down

             list[location] = insertItem; //insert the item at the
                                          //specified position

             length++; //increment the length
        }
} //end insertAt
```

# Array-Based Lists (cont'd.)

- Template `insertEnd` and template `removeAt`

```
template <class elemType>
void arrayListType<elemType>::insertEnd(const elemType& insertItem)
{
  if (length >= maxSize) //the list is full
     cerr << "Cannot insert in a full list" << endl;
  else
  {
       list[length] = insertItem; //insert the item at the end
       length++; //increment the length
  }
} //end insertEnd
template <class elemType>
void arrayListType<elemType>::removeAt(int location)
{
  if (location < 0 || location >= length)
     cerr << "The location of the item to be removed "
          << "is out of range" << endl;
  else
  {
     for (int i = location; i < length - 1; i++)
       list[i] = list[i+1];
     length--;
  }
} //end removeAt
```

# Array-Based Lists (cont'd.)

- Template `replaceAt` and template `clearList`

```
template <class elemType>
void arrayListType<elemType>::retrieveAt
                              (int location, elemType& retItem) const
{
   if (location < 0 || location >= length)
      cerr << "The location of the item to be retrieved is "
         << "out of range." << endl;
   else
      retItem = list[location];
} //end retrieveAt

template <class elemType>
void arrayListType<elemType>::replaceAt
                              (int location, const elemType& repItem)
{
   if (location < 0 || location >= length)
      cerr << "The location of the item to be replaced is "
         << "out of range." << endl;
   else
      list[location] = repItem;
} //end replaceAt
template <class elemType>
void arrayListType<elemType>::clearList()
{
   length = 0;
} //end clearList
```

# Array-Based Lists (cont'd.)

- Definition of the constructor and the destructor

```
template <class elemType>
arrayListType<elemType>::arrayListType(int size)
{
   if (size < 0)
   {
      cerr << "The array size must be positive. Creating "
         << "an array of size 100. " << endl;

      maxSize = 100;
   }
   else
      maxSize = size;

   length = 0;
   list = new elemType[maxSize];
   assert(list != NULL);
}

template <class elemType>
arrayListType<elemType>::~arrayListType()
{
   delete [] list;
}
```

# Array-Based Lists (cont’d.)

- Copy constructor
  - Called when object passed as a (value) parameter to a function
  - Called when object declared and initialized using the value of another object of the same type

```
Type1 a;
…
Type1 b = a;
```

  - Copies the data members of the actual object into the corresponding data members of the formal parameter and the object being created (deep copy)

# Array-Based Lists (cont'd.)

- Copy constructor (cont'd.)
  - Definition

```
template <class elemType>
arrayListType<elemType>::arrayListType
                      (const arrayListType<elemType>& otherList)
{
    maxSize = otherList.maxSize;
    length = otherList.length;
    list = new elemType[maxSize]; //create the array
    assert(list != NULL);         //terminate if unable to allocate
                                  //memory space

    for (int j = 0; j < length; j++) //copy otherList
      list [j] = otherList.list[j];
} //end copy constructor
```

# Array-Based Lists (cont'd.)

- Overloading the assignment operator
  - Definition of the function template

```
template <class elemType>
const arrayListType<elemType>& arrayListType<elemType>::operator=
                    (const arrayListType<elemType>& otherList)
{
  if (this != &otherList)     //avoid self-assignment
  {
    delete [] list;
    maxSize = otherList.maxSize;
    length = otherList.length;

    list = new elemType[maxSize]; //create the array
    assert(list != NULL);         //if unable to allocate memory
                                  //space, terminate the program
    for (int i = 0; i < length; i++)
       list[i] = otherList.list[i];
  }

return *this;
}
```

Why the return type is "`const T&`"?

- `&`: to support *assignment chaining* a=b=c, which is right associative, a = (b = c)
- `const`: avoid (a = b) = c

# Array-Based Lists (cont'd.)

- Searching for an element
  - Linear search example: determining if 27 is in the list
  - Definition of the function template

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | |
|---|---|---|---|---|---|---|---|---|---|
| list | 35 | 12 | 27 | 18 | 45 | 16 | 38 | | ... |

**FIGURE 3-32** List of seven elements

```
template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item) const
{
    int loc;
    bool found = false;

    for (loc = 0; loc < length; loc++)
        if (list[loc] == item)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
} //end seqSearch
```

Why not return `bool`?

Time complexity?

Can we do better?

# Array-Based Lists (cont'd.)

- Inserting an element (duplicates not allowed)

```
template <class elemType>
void arrayListType<elemType>::insert(const elemType& insertItem)
{
  int loc;

  if (length == 0) //list is empty
     list[length++] = insertItem; //insert the item and
                                  //increment the length
  else if (length == maxSize)
     cerr << "Cannot insert in a full list." << endl;
  else
  {
     loc = seqSearch(insertItem);
     if (loc == -1)               //the item to be inserted
                                  //does not exist in the list
        list[length++] = insertItem;
     else
        cerr << "the item to be inserted is already in "
             << "the list. No duplicates are allowed." << endl;
  }
} //end insert
```

Time complexity?

# Array-Based Lists (cont'd.)

- Removing an element

Time complexity?

```
template<class elemType>
void arrayListType<elemType>::remove(const elemType& removeItem)
{
  int loc;
  if (length == 0)
     cerr << "Cannot delete from an empty list." << endl;
  else
  {
     loc = seqSearch(removeItem);
     if (loc != -1)
        removeAt(loc);
     else
        cout << "The item to be deleted is not in the list."
             << endl;
  }
} //end remove
```

# Array-Based Lists (cont'd.)

TABLE 3-1 Time complexity of list operations

| Function | Time-complexity |
| --- | --- |
| isEmpty | $O(1)$ |
| isFull | $O(1)$ |
| listSize | $O(1)$ |
| maxListSize | $O(1)$ |
| print | $O(n)$ |
| isItemAtEqual | $O(1)$ |
| insertAt | $O(n)$ |
| insertEnd | $O(1)$ |
| removeAt | $O(n)$ |
| retrieveAt | $O(1)$ |
| replaceAt | $O(1)$ |
| clearList | $O(1)$ |
| constructor | $O(1)$ |
| destructor | $O(1)$ |
| copy constructor | $O(n)$ |
| overloading the assignment operator | $O(n)$ |
| seqSearch | $O(n)$ |
| insert | $O(n)$ |
| remove | $O(n)$ |

# Summary

- Pointers contain memory addresses
  - All pointers must be initialized
  - Static and dynamic variables
- Static and dynamic arrays
- `new` and `delete`
- Virtual functions
  - Enforce run-time binding of functions
  - Virtual destructor
- Deep copy
  - Overloading assignment operator
  - Overriding copy constructor
- Array-based lists
- Template: use generic code

# Self Exercises

- Programming Exercises: 5, 6, 8, 9, 11, 12