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
  - Contains 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;` and `char *ch;`

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

- Address of operator (`&`)
  - Unary operator
  - Returns address of its operand
- Dereferencing operator (`*`)
  - Unary operator
    - Different from binary multiplication operator
  - Also known as indirection operator
  - Refers to object where the pointer points (operand of the `*`)

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
```

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

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

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

- Operator `new`
  - Allocates single variable
  - Allocates 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 pointerVariable;
delete [ ] pointerVariable;
```

  - Memory leak
    - Memory space that cannot be reallocated
  - Dangling pointers
    - Pointer variables containing addresses of deallocated memory spaces
    - Avoid by setting deleted pointers to `NULL` after delete

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

- Operations on pointer variables
  - Operations allowed
    - Assignment, relational operations; some limited arithmetic operations
    - Can assign value of one pointer variable to another pointer variable of the same type
    - Can compare two pointer variables for equality
    - Can add and subtract integer values from pointer variable
  - 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];
```

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

- Array name: a constant pointer
  - Array name value: constant
  - Increment, decrement operations cannot be applied

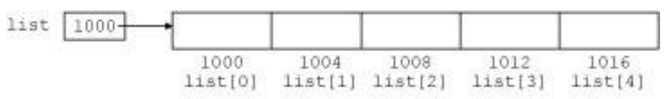


FIGURE 3-14 `list` and array `list`

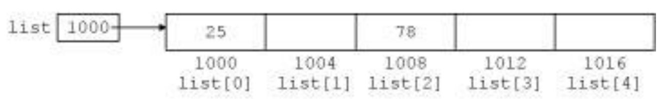


FIGURE 3-15 Array `list` after the execution of the statements `list[0] = 25;` and `list[2] = 78;`

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
  - Declaring a pointer as a value parameter in a function heading
    - Same mechanism used to declare a variable
  - 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.)
  - Declaring formal parameter as reference parameter
    - Must use `&`
    - Between data type name and identifier name, include `*` to make identifier a pointer
    - Between data type name and identifier name, include `&` to make the identifier a reference parameter
    - To make a pointer a reference parameter in a function heading, `*` appears before `&` between data type name and identifier

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

- Functions and pointers (cont'd.)
  - Example

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

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

- Dynamic two-dimensional arrays
  - Creation

```
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 and pointers
  - Pointer arithmetic may create unsuspected or erroneous results
  - Shallow copy
    - Two or more pointers of same type
    - Points to same memory
    - Points to same data

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

- Shallow copy

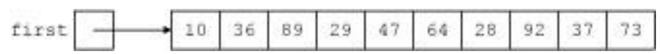


FIGURE 3-16 Pointer `first` and its array

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

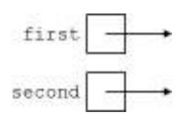


FIGURE 3-18 `first` and `second` after the statement `delete [] second;` executes

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

- Deep copy
  - Two or more pointers have their own data

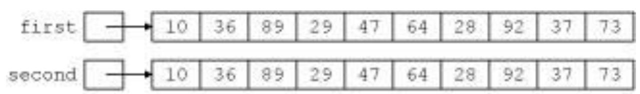


FIGURE 3-19 `first` and `second` both pointing to their own data

Classes and Pointers: Some Peculiarities

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

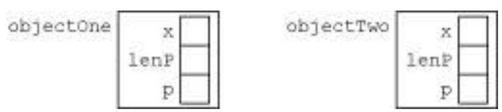


FIGURE 3-20 Objects `objectOne` and `objectTwo`

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

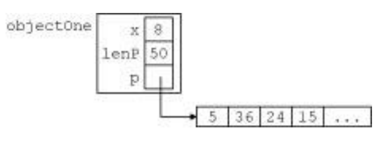


FIGURE 3-21 Object `objectOne` and its data

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

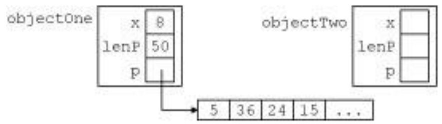


FIGURE 3-22 Objects `objectOne` and `objectTwo`

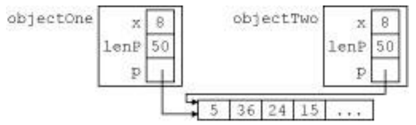


FIGURE 3-23 Objects `objectOne` and `objectTwo` after the statement `objectTwo = objectOne;` executes

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

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

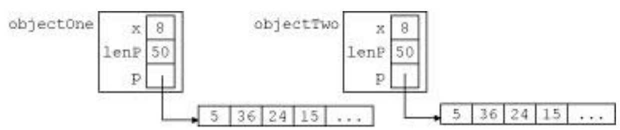


FIGURE 3-24 Objects `objectOne` and `objectTwo`

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
    - Correct by overriding copy constructor definition provided by compiler
  - Syntax to include copy constructor in the definition of a class

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

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

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
    - 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
  - Use a virtual destructor to correct this issue

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 descriptor virtual

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.)
  - Pure virtual functions
  - 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

Array-Based Lists

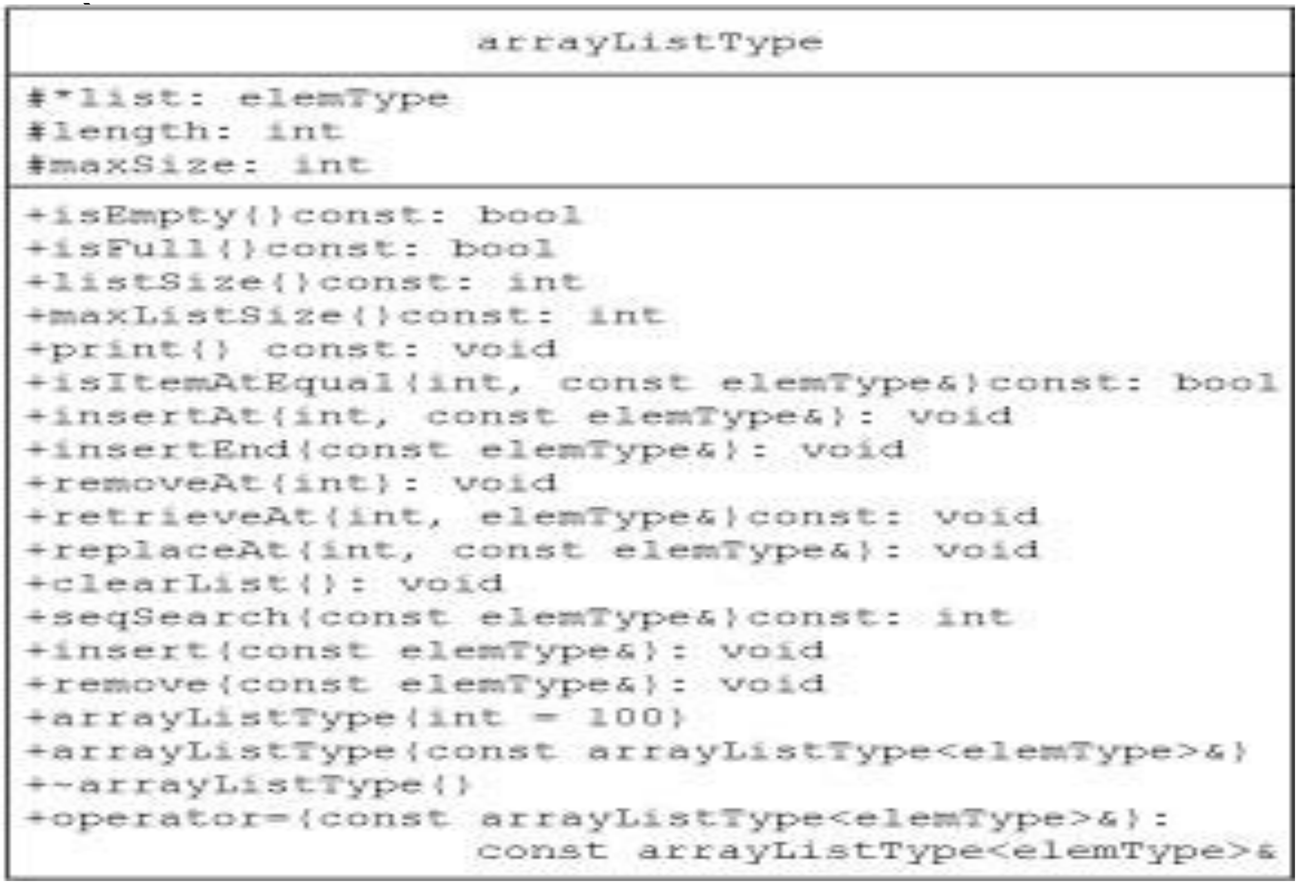
- List
  - Collection of elements of same type
- Length of a list
  - Number of elements in the list
- Many operations may be performed on a list
- Store a list in the computer’s memory
  - Using an array

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 length of the list
    - Number of list elements currently in the array
  - A variable to store array size
    - Maximum number of elements that can be stored in the array
- 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)



```
arrayListType
-----------------------------------------------------------
#*list: elemType
#length: int
#maxSize: int
-----------------------------------------------------------
+isEmpty() const: bool
+isFull() const: bool
+listSize() const: int
+maxListSize() const: int
+print() const: void
+isItemAtEqual(int, const elemType&) const: bool
+insertAt(int, const elemType&): void
+insertEnd(const elemType&): void
+removeAt(int): void
+retrieveAt(int, elemType&) const: void
+replaceAt(int, const elemType&): void
+clearList(): void
+seqSearch(const elemType&) const: int
+insert(const elemType&): void
+remove(const elemType&): void
+arrayListType(int = 100)
+arrayListType(const arrayListType<elemType>&)
+~arrayListType()
+operator=(const arrayListType<elemType>&):
                const arrayListType<elemType>&
```

FIGURE 3-29 UML class diagram of the class `arrayListType`

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
  - Copies the data members of the actual object into the corresponding data members of the formal parameter and the object being created

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;
}
```

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
```

Array-Based Lists (cont'd.)

- Inserting an element

```
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
```

Array-Based Lists (cont'd.)

- Removing an element

```
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	$O(n)$
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
  - Several operators allowed
- Static and dynamic arrays
- Virtual functions
  - Enforce run-time binding of functions
- Array-based lists
  - Several operations allowed
  - Use generic code