Chapter 12: Pointers, Classes, Virtual Functions, Abstract Classes, and Lists
Objectives

In this chapter, you will:

- Use the pointer data type and pointer variables
- Declare and manipulate pointer variables
- Learn about the address of operator and the dereferencing operator
- Discover dynamic variables
- Use the new and delete operators to manipulate dynamic variables
- Learn about pointer arithmetic
Objectives (cont’d.)

• In this chapter, you will (cont’d.)
  – Discover dynamic arrays
  – Explore shallow and deep copies of data
  – Discover the peculiarities of classes with pointer member variables
  – Learn about virtual functions
  – Examine the relationship between the address of operator and classes
  – Become aware of abstract classes
Pointer Data Type and Pointer Variables

- **Pointer variable**: content is a memory address
- No name associated with the pointer data type in C++
Declaring Pointer Variables

• Syntax:

```
dataType *identifier;
```

• Examples:

```
int *p;
char *ch;
```

• These statements are equivalent:

```
int *p;
int* p;
int * p;
```
Declaring Pointer Variables (cont’d.)

• In the statement:
  
  ```
  int* p, q;
  ```
  
  – Only `p` is a pointer variable
  – `q` is an `int` variable

• To avoid confusion, attach the character `*` to the variable name:
  
  ```
  int *p, q;
  ```

  ```
  int *p, *q;
  ```
Address of Operator (&)

• **Address of operator (&):**
  – A unary operator that returns the address of its operand

• **Example:**

```cpp
int x;
int *p;
p = &x;
```

  – Assigns the address of x to p
Dereferencing Operator (*)

- **Dereferencing operator** (or **indirection operator**):
  - When used as a unary operator, * refers to object to which its operand points
- **Example:**
  ```cpp
  cout << *p << endl;
  ```
  - Prints the value stored in the memory location pointed to by p
Classes, structs, and Pointer Variables

• You can declare pointers to other data types:

```cpp
struct studentType
{
    char name[26];
    double gpa;
    int sID;
    char grade;
};

studentType  student;
studentType  *studentPtr;

– student is an object of type studentType
– studentPtr is a pointer variable of type studentType
```
• To store address of `student` in `studentPtr`:
  
  ```
  studentPtr = &student;
  ```

• To store 3.9 in component `gpa` of `student`:
  
  ```
  (*studentPtr).gpa = 3.9;
  ```

  – ( ) used because dot operator has higher precedence than dereferencing operator

  – Alternative: use member access operator arrow (`->`)
Classes, structs, and Pointer Variables (cont’d.)

• Syntax to access a `class (struct)` member using the operator `->`:

  `pointerVariableName->classMemberName`

• Thus,

  `(*studentPtr).gpa = 3.9;`

  is equivalent to:

  `studentPtr->gpa = 3.9;`
Initializing Pointer Variables

• C++ does not automatically initialize variables
• Pointer variables must be initialized if you do not want them to point to anything
  – Initialized using the null pointer: the constant value 0
  – Or, use the NULL named constant
  – The number 0 is the only number that can be directly assigned to a pointer variable
Dynamic Variables

- **Dynamic variables**: created during execution
- C++ creates dynamic variables using pointers
- `new` and `delete` operators: used to create and destroy dynamic variables
  - `new` and `delete` are reserved words in C++
Operator `new`

- `new` has two forms:

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

  - `intExp` is any expression evaluating to a positive integer

- `new` allocates memory (a variable) of the designated type and returns a pointer to it
  - The allocated memory is uninitialized
Operator `new` (cont’d.)

- **Example:**\( p = \text{new \ int};\)
  - Creates a variable during program execution somewhere in memory
  - Stores the address of the allocated memory in \( p \)
- To access allocated memory, use \( *p \)
- A dynamic variable cannot be accessed directly
  - Because it is unnamed
Operator delete (cont’d.)

• **Memory leak**: previously allocated memory that cannot be reallocated
  – To avoid a memory leak, when a dynamic variable is no longer needed, destroy it to deallocate its memory

• `delete` operator: used to destroy dynamic variables

• Syntax:

  ```c++
  delete pointerVariable;       // to deallocate a single
                               // dynamic variable
  delete [] pointerVariable;   // to deallocate a dynamically
                               // created array
  ```
• `delete` only marks the memory space as deallocated
  – Pointer variable may still contain address of deallocated memory space
• If you try to access via the pointer, could result in corrupting data or termination
• Avoid this by setting pointers to `NULL` after delete operation
Operations on Pointer Variables

• `delete` only marks the memory space as deallocated
  – Pointer variable may still contain address of deallocated memory space
• If you try to access via the pointer, could result in corrupting data or termination
• Avoid this by setting pointers to `NULL` after delete operation
Operations on Pointer Variables (cont’d.)

• Pointer arithmetic can be very dangerous:
  – Program can accidentally access memory locations of other variables and change their content without warning
  • Some systems might terminate the program with an appropriate error message

• Always exercise extra care when doing pointer arithmetic
Dynamic Arrays

- **Dynamic array**: array created during program execution

Example:

```cpp
int *p;
p = new int[10];

*p = 25;  // stores 25 into the first memory location
p++;  // to point to next array component
*p = 35;  // stores 35 into the second memory location
```
Dynamic Arrays (cont’d.)

• Can use array notation to access these memory locations

• Example:

   `p[0] = 25;`
   `p[1] = 35;`
   – Stores 25 and 35 into the first and second array components, respectively

• An array name is a constant pointer
Functions and Pointers

• Pointer variable can be passed as a parameter either by value or by reference

• As a reference parameter in a function heading, use &:
  
  ```
  void pointerParameters(int* &p, double *q)
  {
      ...  
  }
  ```
Points and Function Return Values

• A function can return a value of type pointer:

```c
int* testExp(...) {
   . . .
}
```
Dynamic Two-Dimensional Arrays

- You can create dynamic multidimensional arrays
- Examples:

```c++
int *board[4]; // declares board to be an array of four pointers wherein each pointer is of type int

for (int row = 0; row < 4; row++)
    board[row] = new int[6]; // creates the rows of board

int **board; // declares board to be a pointer to a pointer
```
Shallow Versus Deep Copy and Pointers

- **Shallow copy**: when two or more pointers of the same types point to the same memory
  - They point to the same data
  - Danger: deleting one deletes the data pointed to by all of them

- **Deep copy**: when the contents of the memory pointed to by a pointer are copied to the memory location of another pointer
  - Two copies of the data
Shallow Versus Deep Copy and Pointers (cont’d.)

**FIGURE 12-9** Pointer `first` and its array

**FIGURE 12-10** `first` and `second` after the statement `second = first;` executes

**FIGURE 12-11** `first` and `second` after the statement `delete [] second;` executes
Classes and Pointers: Some Peculiarities

- Example class:

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

- Example program statements:

```cpp```
ptrMemberVarType objectOne;
ptrMemberVarType objectTwo;
```cpp```
Destructor

• If `objectOne` goes out of scope, its member variables are destroyed
  – Memory space of dynamic array stays marked as allocated, even though it cannot be accessed

• Solution: in destructor, ensure that when `objectOne` goes out of scope, its array memory is deallocated:

```cpp
ptrMemberVarType::~ptrMemberVarType()
{
    delete [] p;
}
```
Assignment Operator

- After a shallow copy: if `objectTwo.p` deallocates memory space to which it points, `objectOne.p` becomes invalid.

- Solution: extend definition of the assignment operator to avoid shallow copying of data.
Copy Constructor

• Default member-wise initialization:
  • Initializing a class object by using the value of an existing object of the same type

• Example:
  
  `ptrMemberVarType objectThree(objectOne);`

• **Copy constructor**: provided by the compiler
  – Performs this initialization
  – Leads to a shallow copying of the data if class has pointer member variables
Copy Constructor (cont’d.)

- Similar problem occurs when passing objects by value
- Copy constructor automatically executes in three situations:
  - When an object is declared and initialized by using the value of another object
  - When an object is passed by value as a parameter
  - When the return value of a function is an object
Copy Constructor (cont’d.)

• Solution: override the copy constructor

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

• For classes with pointer member variables, three things are normally done:
  – Include the destructor in the class
  – Overload the assignment operator for the class
  – Include the copy constructor
• Can pass an object of a derived class to a formal parameter of the base class type

• **Compile-time binding**: the necessary code to call specific function is generated by compiler
  – Also known as **static binding** or **early binding**

• **Virtual function**: binding occurs at program execution time, not at compile time
  – Declared with reserved word **virtual**
• **Run-time binding:**
  – Compiler does not generate code to call a specific function: it generates information to enable run-time system to generate specific code for the function call
  – Also known as *dynamic* or *late binding*

• *Note*: cannot pass an object of base class type to a formal parameter of the derived class type
Inheritance, Pointers, and Virtual Functions (cont’d.)

• Values of a derived class object can be copied into a base class object

• **Slicing problem**: if derived class has more data members than base class, some data could be lost

• Solution: use pointers for both base and derived class objects
Classes with pointer member variables should have the destructor
   – Destructor should deallocate storage for dynamic objects
• If a derived class object is passed to a formal parameter of the base class type, destructor of the base class executes
   – Regardless of whether object is passed by reference or by value
• Solution: use a virtual destructor (base class)
• Virtual destructor of a base class automatically makes the destructor of a derived class virtual
  – After executing the destructor of the derived class, the destructor of the base class executes
• If a base class contains virtual functions, make the destructor of the base class virtual
Abstract Classes and Pure Virtual Functions

• New classes can be derived through inheritance 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 member functions

• Base class can contain functions that you would want each derived class to implement
  • However, base class may contain functions that may not have meaningful definitions in the base class
• **Pure virtual functions:**
  – Do not have definitions (bodies have no code)

• **Example:** `virtual void draw() = 0;`

• **Abstract class:** a class with one or more virtual functions
  – Can contain instance variables, constructors, and functions that are not pure virtual
  – Class must provide the definitions of constructor/functions that are not pure virtual
Array Based Lists

• **List**: a collection of element of the same type
• Length of a list: the number of elements in a list
• **Array**: an effective way to store a list
• Three variables needed to process a list:
  – `list`: holds the elements
  – `length`: number of elements currently in array
  – `maxSize`: maximum number of elements that can be stored (array size)
Array Based Lists (cont’d.)

```cpp
arrayListType

// Member functions
+isEmpty() const: bool
+isFull() const: bool
+listSize() const: int
+maxListSize() const: int
+print() const: void
+isEqualAt(int, int) const: bool
+insertAt(int, int) = 0: void
+insertEnd(int) = 0: void
+removeAt(int): void
+retrieveAt(int, int&) const: void
+replaceAll(int, int) = 0: void
+clearList(): void
+seqSearch(int) const = 0: int
+remove(int) = 0: void
+arrayListType(int = 100)
+arrayListType(const arrayListType&) = delete
+~arrayListType()
```

**FIGURE 12-22** UML diagram of the class `arrayListType`
Unordered Lists

- unorderedArrayListType class derived from class arrayListType
- Implements insertAt, insertEnd, replaceAt, seqSearch, insert, and remove
- Note that elements of a list need not be distinct
  - Unlike a set, which does not allow duplicates
Ordered Lists

• `orderedArrayListType` class derived from `class arrayListType`

• Elements are in ascending order
& operator can create aliases to an object

Example:

```cpp
int x;
int &y = x;
```

x and y refer to the same memory location

y is like a constant pointer variable

```cpp
y = 25; // sets the value of y (and of x) to 25
x = 2 * x + 30; // updates value of x and y
```
Address of Operator and Classes (cont’d.)

• Address of operator can also be used to return the address of a private member variable of a class
  – However, if you are not careful, this operation can result in serious errors in the program
• Pointer variables contain the addresses of other variables as their values
  • Declare a pointer variable with an asterisk, *, between the data type and the variable
  • Address of operator (&) returns the address of its operand
  • Unary operator * is the dereferencing operator
  • Member access operator (->) accesses the object component pointed to by a pointer
• Dynamic variable: created during execution
  – Created using new, deallocated using delete
• Shallow copy: two or more pointers of the same type point to the same memory
• Deep copy: two or more pointers of the same type have their own copies of the data
• Binding of virtual functions occurs at execution time (dynamic or run-time binding)