Objectives

In this chapter you will:

• Learn about the pointer data type and pointer variables

• Explore how to declare and manipulate pointer variables

• Learn about the address of operator and the dereferencing operator

• Discover dynamic variables
Objectives (continued)

• Explore how to use the **new** and **delete** operators to manipulate dynamic variables
• Learn about pointer arithmetic
• Discover dynamic arrays
• Become aware of the shallow and deep copies of data
• Discover the peculiarities of classes with pointer member variables
Objectives (continued)

- Explore how dynamic arrays are used to process lists
- Learn about virtual functions
- Examine the relationship between the address of operator and classes
- Become aware of abstract classes
Pointer Variables

- **Pointer variable**: content is a memory address

- Declaring Pointer Variables: Syntax

  ```
  dataType *identifier;
  ```

- Examples:

  ```
  int *p;
  char *ch;
  ```
Pointer Variables (continued)

• These statements are equivalent
  
  ```
  int   *p;
  int*  p;
  int   * p;
  ```

• The character `*` can appear anywhere between type name and variable name

• In the statement
  ```
  int*   p, q;
  ```

  only `p` is the pointer variable, not `q`; here `q` is an `int` variable
Pointer Variables (continued)

• To avoid confusion, attach the character * to the variable name

```c
int    *p, q;
```

• The following statement declares both p and q to be pointer variables of the type int

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

- The ampersand, \&, is called the address of operator.
- The address of operator is a unary operator that returns the address of its operand.
Dereferencing Operator (*)

- C++ uses * as the binary multiplication operator and as a unary operator.
- When used as a unary operator, *
  - Called dereferencing operator or indirection operator.
  - Refers to object to which its operand (that is, a pointer) points.
int x = 25;
int *p;
p = &x;    //store the address of x in p

The following statement prints the value stored in the memory space pointed to by p, which is the value of x.

    cout << *p << endl;

The following statement stores 55 in the memory location pointed to by p—that is, in x.

    *p = 55;
int *p;
int num;

FIGURE 14-1  Main memory, p, and num
\texttt{num = 78;}

\textbf{FIGURE 14-2} \texttt{num} after the statement \texttt{num = 78;} executes
$p = &\text{num};$

**Figure 14-3**  $p$ after the statement $p = &\text{num};$ executes
\*p = 24;

**Figure 14-4**  \*p and num after the statement \*p = 24; executes
1. \&p, p, and \*p all have different meanings.
2. \&p means the address of p—that is, 1200 (in Figure 14-4).
3. p means the content of p (1800 in Figure 14-4).
4. \*p means the content (24 in Figure 14-4) of the memory location (1800 in Figure 14-4) pointed to by p (that is, pointed to by the content of memory location 1200).
Example 14-1

```c
int *p;
int x;
```

![Main Memory diagram](image)

**FIGURE 14-5** Main memory, p, and x

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;p</td>
<td>1400</td>
</tr>
<tr>
<td>p</td>
<td>??? (unknown)</td>
</tr>
<tr>
<td>*p</td>
<td>Does not exist (undefined)</td>
</tr>
<tr>
<td>&amp;x</td>
<td>1750</td>
</tr>
<tr>
<td>x</td>
<td>??? (unknown)</td>
</tr>
</tbody>
</table>
\( x = 50; \)

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;p</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>*p</td>
</tr>
<tr>
<td>&amp;x</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

\[ p = \&x; \]

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;p</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>*p</td>
</tr>
<tr>
<td>&amp;x</td>
</tr>
<tr>
<td>x</td>
</tr>
</tbody>
</table>
\texttt{*p = 38;}

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;p</td>
<td>1400</td>
</tr>
<tr>
<td>p</td>
<td>1750</td>
</tr>
<tr>
<td>*p</td>
<td>38</td>
</tr>
<tr>
<td>&amp;x</td>
<td>1750</td>
</tr>
<tr>
<td>x</td>
<td>38</td>
</tr>
</tbody>
</table>
1. A declaration such as

```c
int *p;
```

allocates memory for \( p \) only, not for \( *p \).

2. Assume the following:

```c
int *p;
int x;
```

Then,

a. \( p \) is a pointer variable.

b. The content of \( p \) points only to a memory location of type \textit{int}.

c. Memory location \( x \) exists and is of type \textit{int}. Therefore, the assignment statement

```c
p = &x;
```

is legal. After this assignment statement executes, \( *p \) is valid and meaningful.
Classes, Structs, and Pointer Variables

• You can also declare pointers to other data types, such as classes.

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

```c
studentType    student;
studentType   *studentPtr;
```

• student is an object of type studentType, and studentPtr is a pointer variable of type studentType.
studentPtr = &student;

- This statement stores the address of student in studentPtr.

(*studentPtr).gpa = 3.9;

- This statement stores 3.9 in the component gpa of the object student.
• In C++, the dot operator, ., has a higher precedence than the dereferencing operator.

• In the expression (*studentPtr).gpa, the operator * evaluates first and so the expression *studentPtr evaluates first.

• Because studentPtr is a pointer variable of type studentType, *studentPtr, refers to a memory space of type studentType, which is a struct.

• Therefore, (*studentPtr).gpa refers to the component gpa of that struct.
• Consider the expression \*studentPtr.gpa. Because . (dot) has a higher precedence than \*, the expression studentPtr.gpa evaluates first.

• The expression studentPtr.gpa would result in syntax error as studentPtr is not a struct variable, and so it has no such component as gpa.

• In the expression (*studentPtr).gpa, the parentheses are important.

• The expression (*studentPtr).gpa is a mixture of pointer dereferencing and the class component selection.

• To simplify the accessing of class or struct components via a pointer, C++ provides another operator, called the member access operator arrow, –> . The operator –> consists of two consecutive symbols: a hyphen and the “greater than” sign.
The syntax for accessing a `class (struct)` member using the operator `->` is:

```
pointerVariableName->classMemberName
```

Thus, the statement:

```
(*studentPtr).gpa = 3.9;
```

is equivalent to the statement:

```
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.
- Pointer variables are initialized using the constant value 0, called the **null pointer**.
- The statement `p = 0;` stores the null pointer in `p`; that is, `p` points to nothing. Some programmers use the named constant `NULL` to initialize pointer variables. The following two statements are equivalent:

```c
p = NULL;
p = 0;
```

- 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
- Two operators, `new` and `delete`, to create and destroy dynamic variables
- `new` and `delete` are reserved words
Operator `new`

- The operator `new` has two forms: one to allocate a single variable, and another to allocate an array of variables. The syntax to use the operator `new` is:

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

  where `intExp` is any expression evaluating to a positive integer.

- The operator `new` allocates memory (a variable) of the designated type and returns a pointer to it—that is, the address of this allocated memory. Moreover, the allocated memory is uninitialized.
int *p;
char *q;
int x;

• The statement:
  
  \[ p = \& x; \]
  
  stores the address of \( x \) in \( p \). However, no new memory is allocated.

• Consider the following statement:
  
  \[ p = \text{new int}; \]
  
  This statement creates a variable during program execution somewhere in memory, and stores the address of the allocated memory in \( p \).
  
  The allocated memory is accessed via pointer dereferencing—namely, \( *p \).
int *p;       //p is a pointer of type int
char *name;   //name is a pointer of type char
string *str;  //str is a pointer of type string

p = new int;  //allocates memory of type int
              //and stores the address of the
              //allocated memory in p
*p = 28;      //stores 28 in the allocated memory

name = new char[5]; //allocates memory for an array of
                    //five components of type char and
                    //stores the base address of the array
                    //in name

strcpy(name, "John"); //stores John in name

str = new string; //allocates memory of type string
                   //and stores the address of the
                   //allocated memory in str
*str = "Sunny Day"; //stores the string "Sunny Day" in
                   //the memory pointed to by str
• The operator `new` allocates memory space of a specific type and returns the (starting) address of the allocated memory space.

• If the operator `new` is unable to allocate the required memory space, (for example, there is not enough memory space), then it throws `bad_alloc` exception and if this exception is not handled, it terminates the program with in error message.
The statement in Line 1 allocates memory space of type `int` and stores the address of the allocated memory space into `p`. Suppose that the address of allocated memory space is 1500.
• The statement in Line 2 stores 54 into the memory space that `p` points to.

![Diagram: p and *p after the execution of *p = 54;](image)

**FIGURE 14-9** `p` and `*p` after the execution of `*p = 54;`

• The statement in Line 3 executes, which allocates a memory space of type `int` and stores the address of the allocated memory space into `p`. Suppose the address of this allocated memory space is 1800.

![Diagram: p after the execution of p = new int;](image)

**FIGURE 14-10** `p` after the execution of `p = new int;`
*p = 73; //Line 4

- The statement in Line 4 stores 73 into the memory space that p points.

**Figure 14-8** p after the execution of *p = 73;
• What happened to the memory space 1500 that \( p \) was pointing to after execution of the statement in Line 1?
• After execution of the statement in Line 3, \( p \) points to the new memory space at location 1800.
• The previous memory space at location 1500 is now inaccessible.
• The memory space 1500 remains as marked allocated. In other words, it cannot be reallocated.
• This is called **memory leak**.
• Imagine what would happen if you execute statements such as Line 1 a few thousand times, or a few million times. There will be a good amount of memory leak. The program might then run out of memory spaces for data manipulation, and eventually result in an abnormal termination of the program.
• How to *avoid* memory leak.
• When a dynamic variable is no longer needed, it can be destroyed; that is, its memory can be deallocated.
• The C++ operator `delete` is used to destroy dynamic variables. The syntax to use the operator `delete` has two forms:

```cpp
delete pointerVariable; //to deallocate a single dynamic variable
delete [] pointerVariable; //to deallocate a dynamically created array
```

delete p;
delete [] name;
delete str;
Operations on Pointer Variables

• **Assignment**: value of one pointer variable can be assigned to another pointer of same type
• **Relational operations**: two pointer variables of same type can be compared for equality, etc.
• Some limited arithmetic operations:
  − Integer values can be added and subtracted from a pointer variable
  − Value of one pointer variable can be subtracted from another pointer variable
int *p, *q;

p = q;

• This statement copies the value of q into p. After this statement executes, both p and q point to the same memory location.

• Any changes made to *p automatically change the value of *q, and vice versa.

p == q

• This expression evaluates to true if p and q have the same value—that is, if they point to the same memory location.

p != q

• This expression evaluates to true if p and q point to different memory location.
• These statements increment the value of \( p \) by 4 bytes because \( p \) is a pointer of type int.

\[
p++;
\text{or} \quad p = p + 1;
\]

\[
q++;\quad \text{chPtr++};
\]

• These statements increment the value of \( q \) by 8 bytes and the value of \( \text{chPtr} \) by 1 byte, respectively.

\[
\text{stdPtr++};
\]

• This statement increments the value of \( \text{stdPtr} \) by 40 bytes.
\( p = p + 2; \)

- This statement increments the value of \( p \) by 8 bytes.

- When an integer is added to a pointer variable, the value of the pointer variable is incremented by the integer times the size of the memory that the pointer is pointing to.

- When an integer is subtracted from a pointer variable, the value of the pointer variable is decremented by the integer times the size of the memory to which the pointer is pointing.
• Pointer arithmetic can be very dangerous.
• Using pointer arithmetic, the program can accidentally access the memory locations of other variables and change their content without warning, leaving the programmer trying to find out what went wrong.
• If a pointer variable tries to access either the memory spaces of other variables or an illegal memory space, some systems might terminate the program with an appropriate error message.
• Always exercise extra care when doing pointer arithmetic.
Dynamic Arrays

• An array created during the execution of a program is called a **dynamic array**. To create a dynamic array, we use the second form of the `new` operator.

• The statement:

```
int *p;
```

declares p to be a pointer variable of type `int`. The statement:

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

allocates 10 contiguous memory locations, each of type `int`, and stores the address of the first memory location into `p`. In other words, the operator `new` creates an array of 10 components of type `int`; it returns the base address of the array, and the assignment operator stores the base address of the array into `p`. 
• The statement:
  
  \[
  *p = 25; \\
  \]
  
  stores 25 into the first memory location.

• The statements:

  \[
  p++;  \quad //p \text{ points to the next array component} \\
  *p = 35; \\
  \]
  
  store 35 into the second memory location.

• By using the increment and decrement operations, you can access the components of the array.

• Of course, after performing a few increment operations, it is possible to lose track of the first array component.
C++ allows us to use array notation to access these memory locations.

The statements:

\[
\begin{align*}
p[0] &= 25; \\
p[1] &= 35;
\end{align*}
\]

store 25 and 35 into the first and second array components, respectively.

The following for loop initializes each array component to 0:

\[
\text{for } (j = 0; j < 10; j++) \\
\quad p[j] = 0;
\]

where \( j \) is an int variable.
• The statement:

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

declares `list` to be an array of 5 components.

**Figure 14-12** list and array list
Because the value of \texttt{list}, which is 1000, is a memory address, \texttt{list} is a pointer variable.

The value stored in \texttt{list}, which is 1000, \textit{cannot be altered during program execution}.

That is, the value of \texttt{list} is \textit{constant}.

Therefore, the increment and decrement operations cannot be applied to \texttt{list}.
• Note the data into the array `list` can be manipulated as before.
• The statement `list[0] = 25;` stores 25 into the first array component.
• The statement `list[3] = 78` stores 78 into the fourth component of `list`.

![Diagram of array list](image)

**Figure 14.13** Array `list` after the execution of the statements `list[0] = 25;` and `list[3] = 78;`
• If $p$ is a pointer variable of type \texttt{int}, then the statement:

\begin{verbatim}
  p = list;
\end{verbatim}

copies the value of \texttt{list}, which is 1000, the base address of the array, into $p$.

• We can perform increment and decrement operations on $p$.
• An \textit{array name} is a \textit{constant pointer}.
Example 14-4

```c++
int *intList;                           //Line 1
int arraySize;                          //Line 2

cout << "Enter array size: ";          //Line 3
cin >> arraySize;                      //Line 4
cout << endl;                          //Line 5

intList = new int[arraySize];           //Line 6
```

- The statement in Line 1 declares `intList` to be a pointer of type `int`.
- The statement in Line 2 declares `arraySize` to be an `int` variable.
- The statement in Line 3 prompts the user to enter the size of the array.
- The statement in Line 4 inputs the array size into the variable `arraySize`.
- The statement in Line 6 creates an array of the size specified by `arraySize`, and the base address of the array is stored in `intList`. 
Functions and Pointers

- A pointer variable can be passed as a parameter either by value or by reference.
- To make a pointer a reference parameter in a function heading, * appears before the & between the data type name and the identifier.

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

Both $p$ and $q$ are pointers.
- A function can return a value of type pointer.
Dynamic Two-Dimensional Arrays

• You can also create dynamic multidimensional arrays.
• Dynamic multidimensional arrays are created similarly.
• There are various ways you can create dynamic dimensional arrays.
This statement declares board to be an array of four pointers wherein each pointer is of type `int`.

`board[0], board[1], board[2], and board[3]` are pointers.

You can now use these pointers to create the rows of board.

Suppose that each row of board has six columns. Then the following `for` loop creates the rows of board.

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

In this `for` loop, board is a two-dimensional array of 4 rows and 6 columns.
This statement declares `board` to be a pointer to a pointer.

- `board` and `*board` are pointers.
- `board` can store the address of a pointer or an array of pointers of type `int`.
- `*board` can store the address of an `int` memory space or an array of `int` values.
board = new int* [10];

- This statement creates an array of 10 pointers of type int and assign the address of that array to board.

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

- This for loop creates the columns of board.
- To access the components of board you can use the array subscripting notation.
Shallow versus Deep Copy and Pointers

```c
int *p;
p = new int;
```

- The first statement declares \( p \) to be a pointer variable of type `int`.
- The second statement allocates memory of type `int`, and the address of the allocated memory is stored in \( p \).

![Diagram showing pointer `p` and the memory it points to](image)

**Figure 14-14** Pointer `p` and the memory to which it points
*p = 87;

FIGURE 14-15  Pointer p with 87 in the memory to which it points
int *first;
int *second;

first = new int[10];

**Figure 14-16** Pointer first and the array to which it points

Suppose

**Figure 14-17** Pointer first and its array
In a shallow copy, two or more pointers of the same type point to the same memory.
second = new int[10];

for (int j = 0; j < 10; j++)
    second[j] = first[j];

**Figure 14-20** first and second both pointing to their own data

- In a **deep copy**, two or more pointers have their own data.
Classes and Pointers: Some Peculiarities

Consider the following class

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

Also consider the following statements

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

**Figure 14-21** Objects objectOne and objectTwo
Destructor

- The object `objectOne` has a pointer member variable `p`.
- Suppose that during program execution the pointer `p` creates a dynamic array.
- When `objectOne` goes out of scope, all the member variables of `objectOne` are destroyed.
- However, `p` created a dynamic array, and dynamic memory must be deallocated using the operator `delete`.
- If the pointer `p` does not use the `delete` operator to deallocate the dynamic array, the memory space of the dynamic array would stay marked as allocated, even though it cannot be accessed.
- How do we ensure that when `p` is destroyed, the dynamic memory created by `p` is also destroyed?
- Suppose that `objectOne` is as shown in Figure 14-22.
FIGURE 14-22  Object objectOne and its data
• We can put the necessary code in the destructor to ensure that when `objectOne` goes out of scope, the memory created by the pointer `p` is deallocated.

• The definition of the destructor for the class `pointerDataClass` is:

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

Suppose that `objectOne` and `objectTwo` are as shown in Figure 14-23

**Figure 14-23** Objects `objectOne` and `objectTwo`
objectTwo = objectOne;

![Diagram showing objects objectOne and objectTwo after the statement objectTwo = objectOne; executes](image)

- If `objectTwo.p` deallocates the memory space to which it points, `objectOne.p` would become invalid.

- To avoid this shallow copying of data for classes with a pointer member variable, C++ allows the programmer to extend the definition of the assignment operator.
Once the assignment operator is properly overloaded, both `objectOne` and `objectTwo` have their own data, as shown in Figure 14-25.
Copy Constructor

```cpp
pointerDataClass objectThree(objectOne);
```

- In this statement, the object `objectThree` is being declared and is also being initialized by using the value of `objectOne`.
- The values of the member variables of `objectOne` are copied into the corresponding member variables of `objectThree`.
- This initialization is called the default member-wise initialization.
- The default member-wise initialization is due to the constructor, called the `copy constructor` (provided by the compiler).
• Just as in the case of the assignment operator, because the class `pointerDataClass` has pointer member variables, this default initialization would lead to a shallow copying of the data, as shown in Figure 14-26. (Assume that `objectOne` is given as before.)

**Figure 14-26** Objects `objectOne` and `objectThree`
• As parameters to a function, class objects can be passed either by reference or by value.

• The class `pointerDataClass` has the destructor, which deallocates the memory space pointed to by `p`. Suppose that `objectOne` is as shown in Figure 14-27.

![Figure 14-27 Object objectOne](image)
The function `destroyList` has a formal value parameter, `paramObject`. Now consider the following statement:

```c
destroyList(objectOne);
```
• Because `objectOne` is passed by value, the member variables of `paramObject` should have their own copy of the data. In particular, `paramObject.p` should have its own memory space.

• If a class has pointer member variables:
  • During object declaration, the initialization of one object using the value of another object would lead to a shallow copying of the data, if the default member-wise copying of data is allowed.
  • If, as a parameter, an object is passed by value and the default member-wise copying of data is allowed, it would lead to a shallow copying of the data.
  • In both cases, to force each object to have its own copy of the data, we must override the definition of the copy constructor provided by the compiler.
  • This is usually done by putting a statement that includes the copy constructor in the definition of the class, and then writing the definition of the copy constructor.
• The copy constructor automatically executes in three situations (the first two are described previously):
  • When an object is declared and initialized by using the value of another object
  • When, as a parameter, an object is passed by value
  • When the return value of a function is an object
As a parameter, a class object can be passed either by value or by reference.

Earlier chapters also said that the types of the actual and formal parameters must match.

In the case of classes, C++ allows the user to pass an object of a derived class to a formal parameter of the base class type.

First, we discuss the case when the formal parameter is either a reference parameter or a pointer.
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;
};
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()
{
    baseClass one(5);       //Line 1
    derivedClass two(3, 15);  //Line 2

    one.print();            //Line 3
    two.print();            //Line 4

    cout << "*** Calling the function "
    << "callPrint ***" << endl;  //Line 5

    callPrint(one);         //Line 6
    callPrint(two);         //Line 7

    return 0;
}
The statement in Line 6 calls the function `callPrint` and passes the object `one` as the parameter; it generates the fifth line of the output.

The statement in Line 7 also calls the function `callPrint`, but passes the object `two` as the parameter; it generates the sixth line of the output.

The output generated by the statements in Lines 6 and 7 shows only the value of `x`, even though in these statements a different class object is passed as a parameter.
• What actually occurred is that for both statements (Lines 6 and 7), the member function `print` of the class `baseClass` was executed.

• This is due to the fact that the binding of the member function `print`, in the body of the function `callPrint`, occurred at compile time.

• Because the formal parameter `p` of the function `callPrint` is of type `baseClass`, for the statement `p.print();`, the compiler associates the function `print` of the class `baseClass`.

• In **compile-time binding**, the necessary code to call a specific function is generated by the compiler. (Compile-time binding is also known as **static binding**.)
• For the statement in Line 7, the actual parameter is of type `derivedClass`.
• When the body of the function `callPrint` executes, logically the `print` function of object `two` should execute, which is not the case.
• During program execution, how does C++ correct this problem of making the call to the appropriate function?
• C++ corrects this problem by providing the mechanism of `virtual` functions.
• The binding of `virtual` functions occurs at program execution time, not at compile time.
• This kind of binding is called `run-time binding`.
• In run-time binding, the compiler does not generate the code to call a specific function. Instead, it generates enough information to enable the run-time system to generate the specific code for the appropriate function call.
• Run-time binding is also known as `dynamic binding`.
• In C++, `virtual` functions are declared using the reserved word `virtual`.
class baseClass
{
public:
    virtual void print(); //virtual function
    baseClass(int u = 0);

private:
    int x;
};

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

private:
    int a;
};
Classes and Virtual Destructors

• Classes with pointer member variables should have the destructor.
• The destructor is automatically executed when the class object goes out of scope.
• If the object creates dynamic objects, the destructor can be designed to deallocate the storage for them.
• If a derived class object is passed to a formal parameter of the base class type, the destructor of the base class executes regardless of whether the derived class object is passed by reference or by value.
• Logically, however, the destructor of the derived class should be executed when the derived class object goes out of scope.
• To correct this problem, the destructor of the base class must be virtual.
The **virtual destructor** of a base class automatically makes the destructor of a derived class virtual.

When a derived class object is passed to a formal parameter of the base class type, then when the object goes out of scope, the destructor of the derived class executes.

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

- Other than enforcing run-time binding of functions, virtual functions also have another use.
- Through inheritance we can derive new classes without designing them from scratch.
- The derived classes, in addition to inheriting the existing members of the base class, can add their own members and also redefine or override public and protected member functions of the base class.
- The base class can contain functions that you would want each derived class to implement.
- There are many scenarios when a class is desired to be served as a base class for a number of derived classes. However, the base class may contain certain functions that may not have meaningful definitions in the base class.
class shape
{
public:
    virtual void draw();
    //Function to draw the shape.

    virtual void move(double x, double y);
    //Function to move the shape at the position
    // (x, y).

    ...

    ...

    ...
};
• The definitions of the functions `draw` and `move` are specific to a particular shape.
• Therefore, each derived class can provide an appropriate definition of these functions.
• The functions `draw` and `move` are `virtual` to enforce run-time binding of these functions.
• The way the definition of the `class` `shape` is written when you write the definition of the functions of the `class` `shape`, you must also write the definitions of the functions `draw` and `move`.
• However, at this point there is no shape to `draw` or `move`.
• Therefore, these function bodies have no code.
• One way to handle this is to make the body of these functions empty.
• This solution would work, but it has another drawback. Once we write the definitions of the functions of the `class` `shape`, then we can create an object of this class.
Because there is no shape to work with, we would like to prevent the user from creating objects of the class shape.

It follows that we would like to do the following two things—to not include the definitions of the functions `draw` and `move`, and to prevent the user from creating objects of the class shape.

Because we do not want to include the definitions of the functions `draw` and `move` of the class shape, we must convert these functions to **pure virtual functions**. In this case, the prototypes of these functions are:

```cpp
virtual void draw() = 0;
virtual void move(double x, double y) = 0;
```
Once a class contains one or more pure \texttt{virtual} functions, then that class is called an \textbf{abstract class}. Thus, the abstract definition of the \texttt{class \ shape} is similar to the following:

\begin{verbatim}
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.

};
\end{verbatim}
• Because an abstract class is not a complete class, as it (or its implementation file) does not contain the definitions of certain functions, you cannot create objects of that class.

• Now suppose that we derive the class rectangle from the class shape. To make rectangle a nonabstract class, so that we can create objects of this class, the class (or its implementation file) must provide the definitions of the pure virtual functions of its base class, which is the class shape.

• Note that in addition to the pure virtual functions, an abstract class can contain instance variables, constructors, and functions that are not pure virtual. However, the abstract class must provide the definitions of constructor and functions that are not pure virtual.
Address of Operator and Classes

• The address of operator can create aliases to an object

• Consider the following statements:

```c++
int x;
int &y = x;
```

- \( x \) and \( y \) refer to the same memory location
- \( y \) is like a constant pointer variable
Address of Operator and Classes (continued)

- The statement $y = 25;$ sets the value of $y$ and hence of $x$ to 25.

- Similarly, the statement $x = 2 * x + 30;$ updates the value of $x$ and hence of $y$. 
Summary

- 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
- & is called the address of operator
- & returns the address of its operand
- Unary operator * is the dereference operator
Summary (continued)

• The member access operator arrow, ->, accesses the component of an object pointed to by a pointer
• Dynamic variable: created during execution
• Operator new creates a dynamic variable
• Operator delete deallocates memory occupied by a dynamic variable
• Dynamic array: created during execution
Summary (continued)

- **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
- **List**: homogeneous collection of elements
- Can pass an object of a derived class to a formal parameter of the base class type
- The binding of virtual functions occurs at execution time, not at compile time, and is called dynamic or run-time binding