the same mechanism as you use to declare a variable. To make a formal parameter be a reference parameter, you use **&** when you declare the formal parameter in the function heading. Therefore, to declare a formal parameter as a reference pointer parameter, between the data type name and the identifier name, you must include ***** to make the identifier a pointer and **&** to make it a reference parameter. The obvious question is: In what order should **&** and ***** appear between the data type name and the identifier to declare a pointer as a reference parameter? In C++, to make a pointer a reference parameter in a function heading, ***** appears before the **&** between the data type name and the identifier. The following example illustrates this concept:

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

In the function **pointerParameters**, both **p** and **q** are pointers. The parameter **p** is a reference parameter; the parameter **q** is a value parameter. Furthermore, the function **pointerParameters** can change the value of ***q**, but not the value of **q**. However, the function **pointerParameters** can change the value of both **p** and ***p**.

Pointers and Function Return Values

In C++, the return type of a function can be a pointer. For example, the return type of the function

```
int* testExp(...)
{
     .
     .
     .
}
```

is a pointer of type **int**.

Dynamic Two-Dimensional Arrays

The beginning of this section discussed how to create dynamic one-dimensional arrays. You can also create dynamic multidimensional arrays. In this section, we discuss how to create dynamic two-dimensional arrays. Dynamic multidimensional arrays are created similarly.

There are various ways you can create dynamic two-dimensional arrays. One way is as follows. Consider the statement:

```
int *board[4];
```

This statement declares **board** to be an array of four pointers wherein each pointer is of type `int`. Because `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**.

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

Note that the expression `new int[6]` creates an array of six components of type `int` and returns the base address of the array. The assignment statement then stores the returned address into `board[row]`. It follows that after the execution of the previous `for` loop, **board** is a two-dimensional array of four rows and six columns.

In the previous `for` loop, if you replace the number 6 with the number 10, then the loop will create a two-dimensional array of four rows and 10 columns. In other words, the number of columns of **board** can be specified during execution. However, the way **board** is declared, the number of rows is fixed. So in reality, **board** is not a true dynamic two-dimensional array.

Next, consider the following statement:

```
int **board;
```

This statement declares **board** to be a pointer to a pointer. In other words, **board** and ***board** are pointers. Now **board** can store the address of a pointer or an array of pointers of type `int`, and ***board** can store the address of an `int` memory space or an array of `int` values.

Suppose that you want **board** to be an array of 10 rows and 15 columns. To accomplish this, first we create an array of 10 pointers of type `int` and assign the address of that array to **board**. The following statement accomplishes this:

```
board = new int* [10]; //create an array of 10 int pointers
```

Because the elements of board are `int` pointers, each of them can point to an array of `int` values.

Next, we create the columns of **board**. The following `for` loop accomplishes this:

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

To access the components of **board**, you can use the array subscripting notation discussed in Chapter 8.

Note that the number of rows and the number of columns of **board** can be specified during program execution. The following program further explains how to create two-dimensional arrays.

EXAMPLE 12-7

```
#include <iostream>                                 //Line 1
#include <iomanip>                                  //Line 2

using namespace std;                                //Line 3

void fill(int **p, int rowSize, int columnSize);    //Line 4
void print(int **p, int rowSize, int columnSize);   //Line 5

int main()                                          //Line 6
{                                                   //Line 7
    int **board;                                    //Line 8

    int rows;                                       //Line 9
    int columns;                                    //Line 10

    cout << "Line 11: Enter the number of rows "
         <<"and columns: ";                         //Line 11
    cin >> rows >> columns;                         //Line 12
    cout << endl;                                   //Line 13

        //Create the rows of board
    board = new int* [rows];                        //Line 14

        //Create the columns of board
    for (int row = 0; row < rows; row++)            //Line 15
        board[row] = new int[columns];              //Line 16

        //Insert elements into board
    fill(board, rows, columns);                     //Line 17

    cout << "Line 18: Board:" << endl;              //Line 18

        //Output the elements of board
    print(board, rows, columns);                    //Line 19

    return 0;                                       //Line 20
}                                                   //Line 21

void fill(int **p, int rowSize, int columnSize)
{
    for (int row = 0; row < rowSize; row++)
    {
        cout << "Enter " << columnSize << " number(s)"
            << " for row number " << row << ": ";
        for (int col = 0; col < columnSize; col++)
            cin >> p[row][col];
        cout << endl;
    }
}
```

```
void print(int **p, int rowSize, int columnSize)
{
    for (int row = 0; row < rowSize; row++)
    {
         for (int col = 0; col < columnSize; col++)
            cout << setw(5) << p[row][col];
         cout << endl;
    }
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 11: Enter the number of rows and columns: 3 4

Enter 4 number(s) for row number 0: 1 2 3 4

Enter 4 number(s) for row number 1: 5 6 7 8

Enter 4 number(s) for row number 2: 9 10 11 12

Line 18: Board:
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

The preceding program contains the functions `fill` and `print`. The function `fill` prompts the user to enter the elements of a two-dimensional array of type `int`. The function `print` outputs the elements of a two-dimensional array of type `int`.

For the most part, the preceding output should be clear. Let us look at the statements in the function `main`. The statement in Line 8 declares `board` to be a pointer to a pointer of type `int`. The statements in Lines 9 and 10 declare `int` variables `rows` and `columns`. The statement in Line 11 prompts the user to input the number of rows and number of columns. The statement in Line 12 stores the number of rows in the variable `rows` and the number of columns in the variable `columns`. The statement in Line 14 creates the rows of `board`, and the `for` loop in Lines 15 and 16 creates the columns of `board`. The statement in Line 17 uses the function `fill` to fill the array `board`, and the statement in Line 19 uses the function `print` to output the elements of `board`.

12

Shallow versus Deep Copy and Pointers

In an earlier section, we discussed pointer arithmetic and explained that if we are not careful, one pointer might access the data of another (completely unrelated) pointer. This event might result in unsuspected or erroneous results. Here, we discuss another peculiarity of pointers. To facilitate the discussion, we will use diagrams to show pointers and their related memory.

Consider the following statements:

```
int *first;
int *second;

first = new int[10];
```