# Chapter 6: Pointer Data Type

The pointer data type is unique among the FreeBasic numeric data types. Instead of containing data, like the other numeric types, a pointer contains the memory address of data. On a 32-bit system, the pointer data type is 4 bytes. FreeBasic uses pointers for a number of functions such as Image Create, and pointers are used heavily in external libraries such as the Windows API. Pointers are also quite fast, since the compiler can directly access the memory location that a pointer points to. A proper understanding of pointers is essential to effective programming in FreeBasic.

For many beginning programmers, pointers seem like a strange and mysterious beast. However, if you keep one rule in mind, you should not have any problems using pointers in your program. The rule is very simple: a pointer contains an address, not data. If you keep this simple rule in mind, you should have no problems using pointers.

## Pointers and Memory

You can think of the memory in your computer as a set of post office boxes (P.O. Box) at your local post office. When you go in to rent a P.O. Box, the clerk will give you a number, such as 100. This is the address of your P.O. Box. You decide to write the number down on a slip of paper and put it in your wallet. The next day you go to the post office and pull out the slip of paper. You locate box 100 and look inside the box and find a nice stack of junk mail. Of course, you want to toss the junk mail, but there isn't a trash can handy, so you decide to just put the mail back in the box and toss it later. Working with pointers in FreeBasic is very similar to using a P.O. Box.

When you declare a pointer, it isn't pointing to anything which is analogous to the blank slip of paper. In order to use a pointer, it must be initialized to a memory address,which is the same as writing down the number 100 on the slip of paper. Once you have the address, find the right P.O. Box, you can dereference the pointer, open the mail box, to add or retrieve data from the pointed-to memory location. As you can see there are three basic steps to using pointers.

1. Declare a pointer variable.
2. Initialize the pointer to a memory address.
3. Dereference the pointer to manipulate the data at the pointed-to memory location.

This isn't really any different than using a standard variable, and you use pointers in much the same way as standard variables. The only real difference between the two is that in a standard variable, you can access the data directly, and with a pointer you must dereference the pointer to interact with the data. The following program illustrates the above steps.

'Create a pointer doesn't point to anything.

Dim myPointer As Integer Ptr

'Initialize the pointer to point to 1 integer

myPointer = Callocate(1, Sizeof(Integer))

'Print the address

Print "Pointer address:";myPointer

'Add some meaningful data

\*myPointer = 10

'Print the contents will not be garbage

Print "Memory location contains:";\*myPointer

'Deallocate the pointer

Deallocate myPointer

Sleep

End

Listing 6.1: basicpointer.bas

**Analysis:** First a pointer to an integer is created. The pointer doesn't point to a memory location yet, and if you were to try and use this uninitialized pointer, you would generate an error. The Callocate function is used to set aside memory equal to the size of an integer, that is 4 bytes, and returns the starting address of the memory segment. The second parameter passed to Callocate is the size of the memory unit to allocate. The first parameter is the number of units to allocate. In this program, one unit of size integer is being allocated.

Now that myPointer has been initialized, you can use it. Next the contents of myPointer is printed, which illustrates that the variable contains a memory address and not data. Data is added to the memory location using the indirection operator \*. The indirection operator tells the compiler you want to work with the data that myPointer is pointing to, rather than the memory address contained in the variable. Next the contents of the memory location, is printed, which is now 10.

The memory allocated with Callocate is freed using the Deallocate procedure. The program is closed in the usual way.

Running the program should produce a result similar to the following output.

Pointer address:3089536

Memory location contains: 10

Output 8.1: basicpointer.bas

The address printed on your computer will probably be different, since the operating system allocates the memory used by Callocate.

When your program terminates, all memory that was allocated in the program is freed and returned to the operating system. However, it is good practice to deallocate memory when it is no longer needed, even if it isn't strictly necessary when a program terminates. The better you manage your program's memory, the less chance of problems you will have when running the program on different computer configurations.

## Typed and Untyped Pointers

FreeBasic has two types of pointers, typed and untyped. The preceding program declared a typed pointer, Dim myPointer as Integer Ptr, which tells the compiler that this pointer will be used for integer data. Using typed pointers allows the compiler to do type checking to make sure that you are not using the wrong type of data with the pointer, and simplifies pointer arithmetic.

Untyped pointers are declared using the Any keyword:

Dim myPointer as Any Ptr

Untyped pointers have no type checking and default to size of byte. Untyped pointers are used in the C Runtime Library and many third party libraries, such as the Win32 API, to accommodate the void pointer type in C. You should use typed pointers so that the compiler can check the pointer assignments, unless working with libraries that require the void pointer.

## Pointer Operators

There are two pointer operators in FreeBasic; the indirection operator and the address of operator.

|  |  |  |
| --- | --- | --- |
| Operator | Syntax | Comment |
| \* (Indirection)  [ ] (Index Access) | B = \*myPointer  \*myPointer = B  B = myPointer[index]  myPointer[index] = B | You can access the data in a pointer memory location by either using the indirection operator or using index access. The index format uses the size of the data type to determine the proper indexing. |
| @ (AddressOf) | myPointer = @myVar  myPointer = @mySub()  myPointer = @myFunction() | Returns the memory address of a variable,  subroutine or function. |

Table 6.1: Pointer Operators

You will notice that the address of operator not only returns the memory address of a variable, but it can also return the address of a subroutine or function. You would use the address of a subroutine or function to create a callback function such as used in the CRT function qsort. Callback functions will discussed later in this chapter.

## Memory Functions

FreeBasic has a number of memory allocation functions that are used with pointers, as shown in the following table.

| Function | Syntax | Comment |
| --- | --- | --- |
| Allocate | myPointer = Allocate(number\_of\_bytes) | Allocates number\_of\_bytes and returns the memory address. If myPointer is 0, the memory could not be allocated. The allocated memory segment is not cleared and contains undefined data. |
| Callocate | myPointer = Callocate(number\_of\_elements, size\_of\_elements). | Callocate allocates number\_of\_elements that have size\_of\_elements and returns the memory address. If the memory could not be allocated, Callocate will return 0. The memory segment allocated is cleared |
| Deallocate | Deallocate myPointer | Frees the memory segment pointed to by myPointer. |
| Reallocate | myPointer = Reallocate(pointer, number\_of\_bytes) | Reallocate changes the sizeof a memory segment created with Allocate or Callocate. If the new size is larger than the existing memory segment, the contents of the memory segment remained unchanged. If the new size is smaller, the contents of the memory segment are truncated. If pointer is 0, Reallocate behaves just like Allocate. A 0 is returned if the memory segment cannot be changed. |

Table 6.2: FreeBasic Memory Functions

These functions are useful for creating a number of dynamic structures such as linked lists, ragged or dynamic arrays and buffers used with third party libraries.

When using the Allocate function you must specify the storage size based on the data type using the equation number\_of\_elements \* Sizeof(data type). To allocate space for 10 integers your code would look like this:

myPointer = Allocate(10 \* Sizeof(Integer)).

An integer is 4 bytes so allocating 10 integers will set aside 40 bytes of memory. Allocate does not clear the memory segment, so any data in the segment will be random, meaningless data until it is initialized.

Callocate works in the same fashion, except that the calculation is done internally. To allocate the same 10 integers using Callocate your code would look like this:

myPointer = Callocate(10, Sizeof(Integer))

Unlike Allocate, Callocate will clear the memory segment.

Reallocate will change the size of an existing memory segment, making it larger or smaller as needed. If the new segment is larger than the existing segment, then the data in the existing segment will be preserved. If the new segment is smaller than the existing segment, the data in the existing segment will be truncated. Reallocate does not clear the added memory or change any existing data.

All of these functions will return a memory address if successful. If the functions cannot allocate the memory segment, then a NULL pointer (0) is returned. You should check the return value each time you use these functions to be sure that the memory segment was successfully created. Trying to use a bad pointer will result in undesirable behavior or system crashes.

There is no intrinsic method for determining the size of an allocation; you must keep track of this information yourself.

Caution Be careful not to use the same pointer variable to allocate two or more memory segments. Reusing a pointer without first deallocating the segment it points to will result in the memory segment being lost causing a memory leak.

## Pointer Arithmetic and Pointer Indexing

When you create a memory segment using the allocation functions, you will need away to access the data contained within the segment. In FreeBasic there are two methods for accessing data in the segment; using the indirection operator with pointer arithmetic, and pointer indexing.

Pointer arithmetic, as the name suggests, adds and subtracts values to a pointer to access individual elements within a memory segment. When you create a typed pointer such as Dim myPointer as Integer ptr, the compiler knows that the data being used with this pointer is of size Integer or 4 bytes. The pointer, when initialized, points to the first element of the segment. You can express this as \*(myPtr + 0). To access the second element, you need to add 1 to the pointer, which can be expressed as \*(myPtr + 1). Since the compiler knows that the pointer is an Integer pointer, adding 1 to the pointer reference will actually increment the address contained in myPtr by 4, the size of an Integer. This is why using typed pointers is preferable over untyped pointers. The compiler does much of the work for you in accessing the data in the memory segment.

Notice that the construct is \*(myPtr + 1) and not \*myPtr + 1. The \* operator has higher precedence than +, so \*myPtr + 1 will actually increment the contents myPtr points to, and not the pointer address. \*myPtr will be evaluated first, which returns the contents of the memory location and then +1 will be evaluated, adding 1 to the memory location. By wrapping myPtr + 1 within parenthesis, you force the compiler to evaluate myPtr + 1 first, which increments the pointer address, and then the \* is applied to return the contents of the new address.

Pointer indexing works the same way as pointer arithmetic, but the details are handled by the compiler. \*(myPtr + 1) is equivalent to myPtr[1]. Again, since the compiler knows that myPtr is an integer pointer, it can calculate the correct memory offsets to return the proper values using the index. Which format you use if up to you, most programmers use the index method because of its simplicity.

The following program shows both methods of accessing a memory segment.

Dim myPtr As Integer Ptr

Dim As Integer i

'Try and allocate space for 10 integers

myPtr = Callocate(10, Sizeof(Integer))

'Make sure the space was allocated

If myPtr = 0 Then

Print "Could not allocate space for buffer."

End 1

End If

'Load data into the buffer

Print "Loading data, print data using \*..."

For i = 0 To 9

\*(myPtr + i) = i

Print "Index:";i;" data:";\*(myPtr + i)

Next

Print

'Print data from buffer

Print "Show data using indexing..."

For i = 0 To 9

Print "Index:";i;" data:";myPtr[i]

Next

'Free the memory

Deallocate myPtr

Sleep

End

Listing 6.2: ptraccess.bas

Analysis: The working variables are declared first. Space for 10 integers is created using the Callocate function. The next four lines check to make sure that the memory was allocated. If it wasn't, the program ends. The End 1 terminates the program with an exit code of 1. This is useful for instances where the program may be run from a batch file and you want to make sure the program ran successfully. You can check the exit code in the batch file and take the appropriate action.

First the memory segment is printed using the indirection operator and pointer arithmetic. Next the same thing is done using the index method. Notice that the index method is much more compact and easier to read. The buffer is deallocated, even though it isn't strictly necessary as the program is terminating. Deallocating memory is a good habit to get into, even when it may not be strictly necessary. The program is closed in the usual way.

When you run the program you should see the following output.

Loading data, print data using \*...

Index: 0 data: 0

Index: 1 data: 1

Index: 2 data: 2

Index: 3 data: 3

Index: 4 data: 4

Index: 5 data: 5

Index: 6 data: 6

Index: 7 data: 7

Index: 8 data: 8

Index: 9 data: 9

Show data using indexing...

Index: 0 data: 0

Index: 1 data: 1

Index: 2 data: 2

Index: 3 data: 3

Index: 4 data: 4

Index: 5 data: 5

Index: 6 data: 6

Index: 7 data: 7

Index: 8 data: 8

Index: 9 data: 9

Output 6.2: ptraccess.bas

As you can see from the output, both formats produce the same results, but the index method is a lot easier to read and understand, and less error-prone than the indirection method.

# Pointer Functions

Freebasic has a set of pointer functions to complement the pointer operators. The following table lists the pointer functions.

| Function | Syntax | Comment |
| --- | --- | --- |
| Cptr | myPtr = Cptr(data\_type, expression) | Converts expression to a data\_type pointer. Expression can be another pointer or an integer. |
| Peek | B = Peek(data\_type, pointer) | Peek returns the contents of memory location pointer to by pointer. Data\_type specifies the type of expected data. |
| Poke | Poke data\_type, pointer, expression | Puts the value of expression into the memory location pointed to by pointer. The data\_type specifies the type of data being placed into the memory location. |
| Sadd | myPtr =Sadd(string\_variable) | Returns the location in memory where the string data in a dynamic string is located. |
| Strptr | myPtr =Strptr(string\_variable) | The same as Sadd. |
| Procptr | myPtr = Procptr(function) | Returns the address of a function. This works the same way as the address of operator @. |
| Varptr | myPtr = Varptr(variable) | This function works the same way as the address of operator @. |

Table 8.3: Pointer Functions

The Sadd and Strptr functions will be discussed in the chapter on the string data types. The Peek and Poke functions have been added for the purposes of supporting legacy code. Procptr and Varptr both work just like the address of operator, but Proptr only works on subroutines and functions and Varptr only works on variables. Cptr is useful for casting an untyped pointer to a typed pointer, such as a return value from a third party library.

## Subroutine and Function Pointers

Subroutines and functions, like variables, reside in memory and have an address associated with their entry point. You can use these addresses to create events in your programs, to create pseudo-objects and are used in callback functions. You create a sub or function pointer just like any other pointer except you declare your variable as a pointer to a subroutine or function rather than as a pointer to a data type. Before using a function pointer, it must be initialized to the address of a subroutine or function using Procptr or @. Once initialized, you use the pointer in the same manner as calling the original subroutine or function. The following program illustrates declaring an using a function pointer.

'Declare our function to be used with pointer

Declare Function Power(number As Integer, pwr As Integer) As Integer

'Dim a function pointer

Dim FuncPtr As Function(x As Integer, y As Integer) As Integer

'Get the address of the function

FuncPtr = @Power

'Use the function pointer

Print "2 raised to the power of 4 is";FuncPtr(2, 4)

Sleep

End

'Write the function that will be called

Function Power(number As Integer, pwr As Integer) As Integer

Return number^pwr

End Function

Listing 8.3: funcptr.bas

Analysis: Line 4 declares the function prototype that will be used with the function pointer. Line 7 declares the function pointer using the the As Function syntax. Notice that the Dim statement does not use the Ptr keyword; the compiler knows that this will be a function pointer since it is declared using the As Function method. When declaring a function pointer, the parameter list must match the number and type of parameters of the pointed-to function, but as you can see, the names do not have to match. In fact, the pointer can be declared as Dim FuncPtr As Function(As Integer, As Integer) As Integer, without the parameter names. The only requirement is to make sure that the type and number of parameters, and the return type, match the function declaration and definition. Line 10 initializes the function pointer to the address of the function using the address of operator @. You could use Procptr here as well. Line 13 uses the pointer to call the function. The calling syntax is the same as using the function name: FuncPtr(2,4) is equivalent to Power(2, 4). Lines 15 and 16 close the program in the usual way. Lines 19through 21 define the actual Power function.

Running the program will produce the following result.

2 raised to the power of 4 is 16

Listing 8.4: Output of funcptr.bas

While this example program may not seem to have any advantages over just calling the function directly, you can use this method to call several functions using a single function pointer. For example, if you were creating your own user interface, you could implement events using a function pointer that called one of several different subroutines depending on the object receiving the event. The only requirement would be that each subroutine must contain the same number and type of parameters.

## Creating a Callback Function

One of the primary uses for function pointers is to create callback functions. A callback function is a function that you have created in your program that is called by a function or subroutine in an external library. Windows uses callback functions to enumerate through Window objects like fonts, printers and forms. The qsort, function contained within the C Runtime Library sorts the elements of an array using a callback function to determine the sort order. The prototype for the qsort function is contained instdlib.bi:

declare sub qsort cdecl alias "qsort" (byval as any ptr, byval as size\_t, byval as size\_t, byval as function cdecl(byval as any ptr, byval as any ptr) as integer)

The following lists the parameter information for the qsort subroutine.

1. The first parameter is the address to the first element of the array. The easiest way to pass this information to qsort is to append the address of operator to the first element index: @myArray(0).
2. The second parameter is the number of elements in the array, that is the array count.
3. The third parameter is the size of each element in bytes. For an array of integers, the element size would be 4 bytes.
4. The fourth parameter is a function pointer to the user created compare function. The function must be declared using the Cdecl passing model, as shown in this parameter.

Using this information, you can see how qsort works. By passing the address of the first element along with the count of elements, and the size of each element, qsort can iterate through the array using pointer arithmetic. Qsort will take two array elements, pass them to your user defined compare function and use the compare function's return value to sort the array elements. It does this repeatedly until each array element is in sorted order. The following program uses the qsort subroutine and a compare function to sort an array of integers.

#include "crt.bi"

'Declare the compare function

'This is defined in the same manner as the qsort declaration

Declare Function QCompare Cdecl (Byval e1 As Any Ptr, Byval e2 As Any Ptr) \_

As Integer

'Dimension the array to sort

Dim myArray(10) As Integer

Dim i As Integer

'Seed the random number generator

Randomize Timer

Print "Unsorted"

'Load the array with some random numbers

For i = 0 To 9

'Rnd returns a number between 0 and 1

'This converts the number to an integer

myArray(i) = Int(Rnd \* 20)

'Print unsorted array

Print "i = ";i;" value = ";myArray(i)

Next

Print

'Call the qsort subroutine

qsort @myArray(0), 10, Sizeof(Integer), @QCompare

Print

'Print sorted array.

Print "Sorted"

For i = 0 To 9

'Rnd returns a number between 0 and 1 to convert to integer

Print "i = ";i;" value = ";myArray(i)

Next

Sleep

End

'The qsort function expects three numbers 'from the compare function:

'-1: if e1 is less than e2

'0: if e1 is equal to e2

'1: if e1 is greater than e2

Function QCompare Cdecl (Byval e1 As Any Ptr, \_

Byval e2 As Any Ptr) As Integer

Dim As Integer el1, el2

Static cnt As Integer

'Get the call count and items passed

cnt += 1

'Get the values, must cast to integer ptr

el1 = \*(Cptr(Integer Ptr, e1))

el2 = \*(Cptr(Integer Ptr, e2))

Print "Qsort called";cnt;" time(s) with";el1;" and";el2;"."

'Compare the values

If el1 < el2 Then

Return 1

Elseif el1 > el2 Then

Return 1

Else

Return 0

End If

End Function

Listing 8.5: qsort.bas

Analysis: Line 3 includes the crt.bi file so that the qsort routine will be available in the program. You need to include this file if you want to use any of the CRT functions. Line 7 declares the compare function. You will notice that it is declared as a Cdecl function, which matches the 4th parameter definition in the qsort declaration. Since qsort expects to see either a -1, 0 or 1, the function's return type is an integer. The underscores character \_ at the end of line 7 is the line continuation character and tells the compiler that the following line is actually a part of this line. In line 11 an array of integers is dimensioned. The array will have 10 elements, with indexes from 0 to 9. Line 12dimensions a working variable i, that will be used to load and display the array values.

Line 15 seeds the random number generator by using the value from the Timer function. The Timer function returns the number of seconds, as a double-type value, since the computer was started. Lines 19 through 25 initializes the array with some random integers and displays them to the console screen. The code in line 22, Int(Rnd \* 20), uses the Rnd function which returns a double precision number between 0 and 1. That number is multiplied by twenty to produce a double-precision number between 0and 20 which is then converted to an integer value using the Int function. Since Int returns the largest integer less than or equal to the input, the resulting random number swill range from 0 to 19. In line 24 the value of the current array index is printed to the console screen.

In line 29 the qsort subroutine is called. Since the array indexes range from 0 to 9, the address of operator on the zero index in used for the first parameter. The array size is10, so that is used as the second parameter. The array is an integer array, soSizeof(Integer) is used to pass the array element size. The final parameter is the address of the compare function which is passed to qsort using the address of operator. Lines 34through 37 print out the now sorted array. Lines 39 and 40 close the program in the usual way.

Lines 47 through 66 contain the compare function code. In line 47 the function is defined just like the declare statement in line 7 using two Any Ptr parameters and returning an integer. The Any Ptrs allow qsort to be able to sort any type of data, including composite types. Lines 49 and 50 delcare the function's working variables. Qsort will pass pointers to the functions, not the actual data, so two integers need to bedeclared so that the data that the pointers point to can be compared. The variable cnt is defined as Static so that its value will be preserved between calls to the function. Cnt is used to keep track of how many calls to the function qsort will make as it sorts the array. Lines 55 and 56 use the indirection operator to return the data from the passed pointers. Notice the Cptr is used to convert the Any Ptr parameters to Integer Ptrs before using the indirection operator. Remember that the code inside the parenthesis will be executed before applying the indirection operator. Line 57 prints out the current call count and the passed parameter values.

Line 59 through 65 compare the two values and return the appropriate indicator value back to qsort. The Return statement, as expected, is used to set the function's return value. The function is closed using the End Function keywords in line 66.

When you run the program you should something similar to the following.

Qsort called 23 time(s) with 0 and 7.

Qsort called 24 time(s) with 14 and 0.

Qsort called 25 time(s) with 13 and 13.

Qsort called 26 time(s) with 3 and 13.

Qsort called 27 time(s) with 10 and 3.

Qsort called 28 time(s) with 7 and 10.

Qsort called 29 time(s) with 0 and 7.

Qsort called 30 time(s) with 13 and 13.

Qsort called 31 time(s) with 3 and 13.

Qsort called 32 time(s) with 10 and 3.

Qsort called 33 time(s) with 7 and 10.

Qsort called 34 time(s) with 13 and 13.

Qsort called 35 time(s) with 3 and 13.

Qsort called 36 time(s) with 10 and 3.

Qsort called 37 time(s) with 13 and 13.

Qsort called 38 time(s) with 3 and 13.

Qsort called 39 time(s) with 13 and 13.

Sorted

i = 0 value = 18

i = 1 value = 8

i = 2 value = 13

i = 3 value = 13

i = 4 value = 3

i = 5 value = 10

i = 6 value = 7

i = 7 value = 0

i = 8 value = 14

i = 9 value = 2

Output 8.3: Output of qsort.bas

The first group of numbers show the unsorted array. The middle group of numbers show the number of times the compare function is called along with the values being sorted. The last group of numbers show the sorted array. Even though qsort is called quite a number of times even on this small array, the routine is extremely fast since it uses pointers to sort the values in the array.

Pointer to Pointer

In FreeBasic you can create a pointer to any of the supported data types, including the pointer data type. A pointer to a pointer is useful in situations where you need to return a pointer to a function or in creating specialized data structures such as linked-lists and ragged arrays. A pointer to a pointer is called multi-level indirection.

Caution You can have as many levels of indirection as needed, but anything beyond two levels is rarely useful and difficult to manage.

One application of a pointer to pointer is the creation of memory arrays. The following program demonstrates the creation, manipulation and freeing of a memory

array.

'Declare a pointer to an int pointer

Dim myMemArray As Integer Ptr Ptr

Dim As Integer i, j

'Create 10 rows of integer pointers

myMemArray = Callocate(10, Sizeof(Integer Ptr))

'Add 10 columns of integers to each row

For i = 0 To 9

myMemArray[i] = Callocate(10, Sizeof(Integer))

Next

'Add some data to the memory segment

For i = 0 To 9

For j = 0 To 9

myMemArray[i][j] = Int(Rnd \* 10)

Next

Next

'Print out data

For i = 0 To 9

For j = 0 To 9

Print "i,j = ";i;",";j;" Mem Array =";myMemArray[i][j]

Next

Next

'Free memory segment

For i = 0 To 9

Deallocate myMemArray[i]

Next

'Free the pointer to pointer

DeAllocate myMemArray

Sleep

End

Listing 8.6: memarray.bas

Analysis: Line 4 dimensions a pointer to an integer pointer, myMemArray, which will simulate a two dimensional array, with myMemArray pointing to a list of integer

pointers. This list of pointers will comprise the “rows” of the array. Line 5 just declares some working variables that are used in the For-Next loops later in the program.

Line 8 creates the rows of the array by allocating a memory segment that will contain 4 integer pointers, which are initialized in lines 11 through 13. Remember that the index method of accessing a pointer automatically does the pointer arithmetic for you. The code in lines 11 through 13 iterates through the memory segment created in line 8 and initializes the memory segment with pointers to memory segments that will contain integers. In other words, you have a pointer that is pointing to a list of pointers. These newly created memory segments are the columns for each row. Each column index will be a pointer to a memory location containing an integer.

Lines 16 through 20 add some random numbers to the memory array. Notice that by using the indexing method you can access the memory array just like a normal array. ipoints to the row (the list of pointers) and j points to the individual columns within that row which contain the integer data. The same method is used in lines 23 through 27 to print out the array values.

Lines 30 through 32 free the individual rows of memory that were created in line 8. It is important that each row be freed before freeing myMemArray. If you were to just freemy MemArray, the rows would still be in memory, but unaccessible, causing a memory leak. Once all the rows have been freed, myMemArray can be freed in line 34. Since the program is terminating, Deallocating the memory is not strictly required in this instance, but if you needed to reuse the memory, then you must Deallocate in the method described, otherwise you will get a memory leak while the program is running. Lines 36and 37 close the program in the usual way.

Running the program should produce a result similar to the following.

i,j = 0, 1 Mem Array = 5

i,j = 0, 2 Mem Array = 1

i,j = 0, 3 Mem Array = 8

i,j = 0, 4 Mem Array = 5

i,j = 1, 0 Mem Array = 4

i,j = 1, 1 Mem Array = 3

i,j = 1, 2 Mem Array = 8

i,j = 1, 3 Mem Array = 8

i,j = 1, 4 Mem Array = 7

i,j = 2, 0 Mem Array = 1

i,j = 2, 1 Mem Array = 8

i,j = 2, 2 Mem Array = 7

i,j = 2, 3 Mem Array = 5

i,j = 2, 4 Mem Array = 3

i,j = 3, 0 Mem Array = 0

i,j = 3, 1 Mem Array = 0

i,j = 3, 2 Mem Array = 3

i,j = 3, 3 Mem Array = 1

i,j = 3, 4 Mem Array = 1

i,j = 4, 0 Mem Array = 9

i,j = 4, 1 Mem Array = 4

i,j = 4, 2 Mem Array = 1

i,j = 4, 3 Mem Array = 0

i,j = 4, 4 Mem Array = 0

Output 8.4: Output of memarray.bas

As you can see from the output, the memory array behaves exactly like a predefined array. This structure is useful for adding dynamic arrays to type definitions, which normally cannot hold a dynamic array. You will see this in more detail in the chapter on composite types.

One last note on this program. If you run the program more than once, you will notice that the values in the array are always the same, even though the program is generating random numbers. This is because the program did not seed the random number generator using the Randomize statement. To get different numbers for each run, add Randomize Timer before calling the Rnd function.

A Look Ahead

Pointers are an efficient and powerful data type, and you will be seeing more of them in this book, including the next chapter where you will explore the string data types available in FreeBasic.