Unsafe Code

As you have just seen, C# is very good at hiding much of the basic memory management from the developer, thanks to the garbage collector and the use of references. However, cases exist in which you will want direct access to memory. For example, you might want to access a function in an external (non-.NET) DLL that requires a pointer to be passed as a parameter (as many Windows API functions do), or possibly for performance reasons. This section examines C#’s facilities that provide direct access to the contents of memory.

Pointers

Although we are introducing *pointers* as if they were a new topic, in reality pointers are not new at all. You have been using references freely in your code, and a reference is simply a type-safe pointer. You have already seen how variables that represent objects and arrays actually store the memory address of where the corresponding data (the *referent*) is stored. A pointer is simply a variable that stores the address of something else in the same way as a reference. The difference is that C# does not allow you direct access to the address contained in a reference variable. With a reference, the variable is treated syntactically as if it stores the actual contents of the referent.

C# references are designed to make the language simpler to use and to prevent you from inadvertently doing something that corrupts the contents of memory. With a pointer, on the other hand, the actual memory address is available to you. This gives you a lot of power to perform new kinds of operations. For example, you can add 4 bytes to the address, so that you can examine or even modify whatever data happens to be stored 4 bytes further on in memory.

The two main reasons for using pointers are:

- **Backward compatibility -** Despite all of the facilities provided by the .NET runtime, it is still possible to call native Windows API functions, and for some operations this may be the only way to accomplish your task. These API functions are generally written in C and often require pointers as parameters. However, in many cases it is possible to write the DllImport declaration in a way that avoids use of pointers; for example, by using the System.IntPtr class.

- **Performance -** On those occasions where speed is of the utmost importance, pointers can provide a route to optimized performance. Provided that you know what you are doing, you can ensure that data is accessed or manipulated in the most efficient way. However, be aware that, more often than not, there are other areas of your code where you can make the necessary performance improvements without resorting to using pointers. Try using a code profiler to look for the bottlenecks in your code - one comes with Visual Studio 2005.

Low-level memory access comes at a price. The syntax for using pointers is more complex than that for reference types, and pointers are unquestionably more difficult to use correctly. You need good programming skills and an excellent ability to think carefully and logically about what your code is doing in order to use pointers successfully. If you are not careful, it is very easy to introduce subtle, difficult-to-find bugs into your program using pointers. For example, it is easy to overwrite other variables, cause stack overflows, access areas of memory that don’t store any variables, or even overwrite information about your code that is needed by the .NET runtime, thereby crashing your program.

In addition, if you use pointers your code must be granted a high level of trust by the runtime’s code access security mechanism or it will not be allowed to execute. Under the default code access security policy, this is only possible if your code is running on the local machine. If your code must be run from a remote location, such as the Internet, users must grant your code additional permissions for it to work. Unless the users trust you and your code, they are unlikely to grant these permissions. Code access security is discussed more in Chapter 19, “.NET Security.”

Despite these issues, pointers remain a very powerful and flexible tool in the writing of efficient code and are worth learning about.

Tip We strongly advise against using pointers unnecessarily because your code will not only be harder to write and debug, but it will also fail the memory type-safety chec… er 1, “.NET Architecture.”

Writing Unsafe Code

As a result of the risks associated with pointers, C# allows the use of pointers only in blocks of code that you have specifically marked for this purpose. The keyword to do this is unsafe. You can mark an individual method as being unsafe like this:

```
unsafe int GetSomeNumber() {    // code that can use pointers }
```

Any method can be marked as unsafe, irrespective of what other modifiers have been applied to it (for example, static methods or virtual methods). In the case of methods, the unsafe modifier applies to the method's parameters, allowing you to use pointers as parameters. You can also mark an entire class or struct as unsafe, which means that all of its members are assumed to be unsafe:

```
unsafe class MyClass {    // any method in this class can now use pointers }
```

Similarly, you can mark a member as unsafe:

```
class MyClass {    unsafe int *pX;   // declaration of a pointer field in a class }
```

Or you can mark a block of code within a method as unsafe:

```
void MyMethod() {    // code that doesn't use pointers    unsafe    {       // unsafe code that uses pointers here    }    // more 'safe' code that doesn't use pointers }
```

Note, however, that you cannot mark a local variable by itself as unsafe:

```
int MyMethod() {    unsafe int *pX;   // WRONG }
```

If you want to use an unsafe local variable, you will need to declare and use it inside a method or block that is unsafe. There is one more step before you can use pointers. The C# compiler rejects unsafe code unless you tell it that your code includes unsafe blocks. The flag to do this is unsafe. Hence, to compile a file named MySource.cs that contains unsafe blocks (assuming no other compiler options), the command is:

```
csc /unsafe MySource.cs
```

or:

```
csc –unsafe MySource.cs
```

Tip If you are using Visual Studio 2005, you will find the option to compile unsafe code in the Build tab of the project properties window.

Pointer Syntax

Once you have marked a block of code as unsafe, you can declare a pointer using this syntax:

```
int* pWidth, pHeight; double* pResult; byte*[] pFlags;
```

This code declares four variables: pWidth and pHeight are pointers to integers, pResult is a pointer to a double, and pFlags is an array of pointers to bytes. It is common practice to use the prefix p in front of names of pointer variables to indicate that they are pointers. When used in a variable declaration, the symbol * indicates that you are declaring a pointer (that is, something that stores the address of a variable of the specified type).

Tip C++ developers should be aware of the syntax difference between C++ and C#. The C# statement int* pX, pY; corresponds to the C++ statement int *pX, *pY;. In C#, the * symbol is associated with the type rather than the variable name.

Once you have declared variables of pointer types, you can use them in the same way as normal variables, but first you need to learn two more operators:

- & means *take the address of*, and converts a value data type to a pointer, for example int to *int. This operator is known as the *address operator*.
- * means *get the contents of this address*, and converts a pointer to a value data type (for example, *float to float). This operator is known as the *indirection operator* (or sometimes as the *dereference operator*).

You will see from these definitions that & and * have opposite effects.

Tip You might be wondering how it is pos[illegible]use these symbols also refer to the operators of bitwise AND (&) and mu[illegible]ou and the compiler to know what is

meant in each case, because with the new pointer meanings, these symbols always appear as unary operators - they only act on one variable and appear in front of that variable in your code. On the other hand, bitwise AND and multiplication are binary operators - they require two operands.

The following code shows examples of how to use these operators:

```
int x = 10; int* pX, pY; pX = &x; pY = pX; *pY = 20;
```

You start off by declaring an integer, x, with the value 10 followed by two pointers to integers, pX and pY. You then set pX to point to x (that is, you set the contents of pX to be the address of x). Then you assign the value of pX to pY, so that pY also points to x. Finally, in the statement *pY=20, you assign the value 20 as the contents of the location pointed to by pY - in effect changing x to 20 because pY happens to point to x. Note that there is no particular connection between the variables pY and x. It's just that at the present time, pY happens to point to the memory location at which x is held.

To get a better understanding of what is going on, consider that the integer x is stored at memory locations 0x12F8C4 through 0x12F8C7 (1243332 to 1243335 in decimal) on the stack (there are four locations be cause an int occupies 4 bytes). Because the stack allocates memory downward, this means that the variables pX will be stored at locations 0x12F8C0 to 0x12F8C3, and pY will end up at locations 0x12F8BC to 0x12F8BF. Note that pX and pY also occupy 4 bytes each. That is not because an int occupies 4 bytes. It's because on a 32-bit processor you need 4 bytes to store an address. With these addresses, after executing the previous code, the stack will look like Figure 11-5.

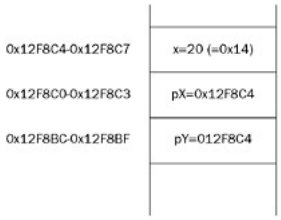

Figure 11-5

Tip Although this process is illustrated with integers, which will be stored consecutively on the stack on a 32-bit processor, this doesn't happen for all data types. The reason is that 32-bit processors work best when retrieving data from memory in 4-byte chunks. Memory on such machines tends to be divided into 4-byte blocks, and each block is sometimes known under Windows as a DWORD because this was the name of a 32-bit unsigned int in pre-.NET days. It is most efficient to grab DWORDs from memory - storing data across DWORD boundaries normally results in a hardware performance hit. For this reason, the .NET runtime normally pads out data types so that the memory they occupy is a multiple of 4. For example, a short occupies 2 bytes, but if a short is placed on the stack, the stack pointer will still be decremented by 4, not 2, so that the next variable to go on the stack will still start at a DWORD boundary.

You can declare a pointer to any value type (that is, any of the predefined types uint, int, byte, and so on, or to a struct). However, it is not possible to declare a pointer to a class or array; this is because doing so could cause problems for the garbage collector. In order to work properly, the garbage collector needs to know exactly what class instances have been created on the heap, and where they are, but if your code started manipulating classes using pointers, you could very easily corrupt the information on the heap concerning classes that the .NET runtime maintains for the garbage collector. In this context, any data type that the garbage collector can access is known as a *managed type*. Pointers can only be declared as *unmanaged types* because the garbage collector cannot deal with them.

Casting Pointers to Integer Types

Because a pointer really stores an integer that represents an address, you won't be surprised to know that the address in any pointer can be converted to or from any integer type. Pointer-to-integer-type conversions must be explicit. Implicit conversions are not available for such conversions. For example, it is perfectly legitimate to write the following:

```
int x = 10; int* pX, pY; pX = &x; pY = pX; *pY = 20; uint y = (uint)pX; int* pD = (int*)y;
```

The address held in the pointer pX is cast to a uint and stored in the variable y. You have then cast y back to an int* and stored it in the new variable pD. Hence, now pD also points to the value of x.

The primary reason for casting a pointer value to an integer type is to display it. The Console.Write() and Console.WriteLine() methods do not have any overloads that can take pointers, but will accept and display pointer values that have been cast to integer types:

```
  Console.WriteLine("Address is " + pX);   // wrong -- will give a
// compilation error Console.WriteLine("Address is " + (uint)pX);   // OK
```

You can cast a pointer to any of the integer types. However, because an address occupies 4 bytes on 32-bit systems, casting a pointer to anything other than a uint, long, or ulong is almost certain to lead to overflow errors. (An int causes problems because its range is from roughly –2 billion to 2 billion, whereas an address runs from zero to about 4 billion.) When C# is released for 64-bit processors, an address will occupy 8 bytes. Hence, on such systems, casting a pointer to anything other than ulong is likely to lead to overflow errors. It is also important to be aware that the checked keyword does not apply to conversions involving pointers. For such conversions, exceptions will not be raised when overflows occur, even in a checked context. The .NET runtime assumes that if you are using pointers you know what you are doing and are not worried about possible overflows.

Casting between Pointer Types

You can also explicitly convert between pointers pointing to different types. For example:

```
byte aByte = 8; byte* pByte= &aByte; double* pDouble = (double*)pByte;
```

This is perfectly legal code, though again, if you try something like this, be careful. In this example, if you look at the double value pointed to by pDouble, you will actually be looking up some memory that contains a byte (aByte), combined with some other memory, and treating it as if this area of memory contained a double, which won't give you a meaningful value. However, you might want to convert between types in order to implement the equivalent of a C union, or you might want to cast pointers to other types into pointers to sbyte in order to examine individual bytes of memory.

void Pointers

If you want to maintain a pointer, but do not want to specify what type of data it points to, you can declare it as a pointer to a void:

```
int* pointerToInt; void* pointerToVoid; pointerToVoid = (void*)pointerToInt;
```

The main use of this is if you need to call an API function that requires void* parameters. Within the C# language, there isn't a great deal that you can do using void pointers. In particular, the compiler will flag an error if you attempt to dereference a void pointer using the * operator.

Pointer Arithmetic

It is possible to add or subtract integers to and from pointers. However, the compiler is quite clever about how it arranges for this to be done. For example, suppose that you have a pointer to an int and you try to add 1 to its value. The compiler will assume that you actually mean you want to look at the memory location following the int, and hence will increase the value by 4 bytes - the size of an int. If it is a pointer to a double, adding 1 will actually increase the value of the pointer by 8 bytes, the size of a double. Only if the pointer points to a byte or sbyte (1 byte each) will adding 1 to the value of the pointer actually change its value by 1.

You can use the operators +, -, +=, -=, ++, and -- with pointers, with the variable on the right-hand side of these operators being a long or ulong.

Tip It is not permitted to carry out arithmetic operations on void pointers.

For example, assume these definitions:

```
uint u = 3; byte b = 8; double d = 10.0; uint* pUint= &u;        // size of a uint is 4 byte* pByte = &b;
// size of a byte is 1 double* pDouble = &d;   // size of a double is 8
```

Next, assume the addresses to which these pointers point are:

- pUint: 1243332
- pByte: 1243328
- pDouble: 1243320

Then execute this code:

```
++pUint;              // adds (1*4) = 4 bytes to pUint pByte -= 3;           // subtracts (3*1) = 3 bytes
from pByte double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

The pointers now contain:

- pUint: 1243336

- pByte: 1243325

- pDouble2: 1243352

Important The general rule is that adding a number ***X*** to a pointer to type ***T*** with value ***P*** gives the result ***P*** + ***X****(sizeof(***T***)).

Tip You need to be aware of the previous rule. If successive values of a given type are stored in successive memory locations, pointer addition works very well to allow you to move pointers between memory locations. If you are dealing with types such as byte or c b char, though, whose sizes are not multiples of 4, successive values will not, by default, be stored in successive memory locations.

You can also subtract one pointer from another pointer, provided that both pointers point to the same data type. In this case, the result is a long whose value is given by the difference between the pointer values divided by the size of the type that they represent:

```
  double* pD1 = (double*)1243324;   // note that it is perfectly valid to
// initialize a pointer like this. double* pD2 = (double*)1243300; long L = pD1-pD2;                 //
gives the result 3 (=24/sizeof(double))
```

The sizeof Operator

This section has been referring to the sizes of various data types. If you need to use the size of a type in your code, you can use the sizeof operator, which takes the name of a data type as a parameter and returns the number of bytes occupied by that type. For example:

```
  int x = sizeof(double);
```

This will set x to the value 8.

The advantage of using sizeof is that you don't have to hard-code data type sizes in your code, making your code more portable. For the predefined data types, sizeof returns the following values:

```
 sizeof(sbyte) = 1;              sizeof(byte) = 1; sizeof(short) = 2;              sizeof(ushort) = 2;
sizeof(int) = 4;                sizeof(uint) = 4; sizeof(long) = 8;               sizeof(ulong) = 8;
sizeof(char) = 2;               sizeof(float) = 4; sizeof(double) = 8;             sizeof(bool) = 1;
```

You can also use sizeof for structs that you define yourself, although in that case, the result depends on what fields are in the struct. You cannot use sizeof for classes, and it can only be used in an unsafe code block.

Pointers to Structs: The Pointer Member Access Operator

Pointers to structs work in exactly the same way as pointers to the predefined value types. There is, however, one condition - the struct must not contain any reference types. This is due to the restriction mentioned earlier that pointers cannot point to any reference types. To avoid this, the compiler will flag an error if you create a pointer to any struct that contains any reference types.

Suppose that you had a struct defined like this:

```
  struct MyStruct {    public long X;    public float F; }
```

You could define a pointer to it like this:

```
  MyStruct* pStruct;
```

Then you could initialize it like this:

```
  MyStruct Struct = new MyStruct(); pStruct = &Struct;
```

It is also possible to access member values of a struct through the pointer:

```
  (*pStruct).X = 4; (*pStruct).F = 3.4f;
```

However, this syntax is a bit complex. For this reason, C# defines another operator that allows you to access members of structs through pointers using a simpler syntax. It is known as the *pointer member access operator*, and the symbol is a dash followed by a greater-than sign, so it looks like an arrow: ->.

Tip C++ developers will recognize the pointer member access operator, because C++ uses the same symbol for the same purpose.

Using the pointer member access operat

```
pStruct->X = 4; pStruct->F = 3.4f;
```

You can also directly set up pointers of the appropriate type to point to fields within a struct:

```
long* pL = &(Struct.X); float* pF = &(Struct.F);
```

or

```
long* pL = &(pStruct->X); float* pF = &(pStruct->F);
```

Pointers to Class Members

As indicated earlier, it is not possible to create pointers to classes. That's because the garbage collector does not maintain any information about pointers, only about references, so creating pointers to classes could cause garbage collection to not work properly.

However, most classes do contain value type members, and you might want to create pointers to them. This is possible but requires a special syntax. For example, suppose that you rewrite the struct from the previous example as a class:

```
class MyClass {    public long X;    public float F; }
```

Then you might want to create pointers to its fields, X and F, in the same way as you did earlier. Unfortunately, doing so will produce a compilation error:

```
MyClass myObject = new MyClass(); long* pL = &(myObject.X);   // wrong -- compilation error float* pF = &(myObject.F);  // wrong -- compilation error
```

Although X and F are unmanaged types, they are embedded in an object, which sits on the heap. During garbage collection, the garbage collector might move MyObject to a new location, which would leave pL and pF pointing to the wrong memory addresses. Because of this, the compiler will not let you assign addresses of members of managed types to pointers in this manner.

The solution is to use the fixed keyword, which tells the garbage collector that there may be pointers referencing members of certain objects, so those objects must not be moved. The syntax for using fixed looks like this if you just want to declare one pointer:

```
MyClass myObject = new MyClass(); fixed (long* pObject = &(myObject.X)) {    // do something }
```

You define and initialize the pointer variable in the brackets following the keyword fixed. This pointer variable (pObject in the example) is scoped to the fixed block identified by the curly braces. As a result of doing this, the garbage collector knows not to move the myObject object while the code inside the fixed block is executing.

If you want to declare more than one pointer, you can place multiple fixed statements before the same code block:

```
MyClass myObject = new MyClass(); fixed (long* pX = &(myObject.X)) fixed (float* pF = &(myObject.F)) {
// do something }
```

You can nest entire fixed blocks if you want to fix several pointers for different periods:

```
MyClass myObject = new MyClass(); fixed (long* pX = &(myObject.X)) {    // do something with pX    fixed (float* pF = &(myObject.F))    {       // do something else with pF    } }
```

You can also initialize several variables within the same fixed block, provided that they are of the same type:

```
MyClass myObject = new MyClass(); MyClass myObject2 = new MyClass(); fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X)) {    // etc. }
```

In all these cases, it is immaterial whether the various pointers you are declaring point to fields in the same or different objects or to static fields not associated with any class instance.

Pointer Example: PointerPlayaround

This section presents an example that uses pointers. The following code is an example named PointerPlayaround. It does some simple pointer manipulation and displays the results, allowing you to see what is happening in memory and where variables are stored:

```
using System; namespace Wrox.ProCSharp.Memory {    class MainEntryPoint    {       static unsafe void Main()       {          int x=10;          short y = -1;          byte y2 = 4;          double z = 1.5;
int* pX = &x;          short* pY = &y;          double* pZ = &z;          Console.WriteLine(
"Address of x is 0x{0:X}, size is {1}, value is {2}",             (uint)&x, sizeof(int), x);
Console.WriteLine(             "A                                                  is {2}",             (uint)&y,
sizeof(short), y);          Consc                                                  s 0x{0:X}, size is {1}, value is
```

```
{2}",             (uint)&y2, sizeof(byte), y2);          Console.WriteLine(             "Address of z is
0x{0:X}, size is {1}, value is {2}",             (uint)&z, sizeof(double), z);          Console.WriteLine(
"Address of pX=&x is 0x{0:X}, size is {1}, value is 0x{2:X}",             (uint)&pX, sizeof(int*),
(uint)pX);          Console.WriteLine(             "Address of pY=&y is 0x{0:X}, size is {1}, value is
0x{2:X}",             (uint)&pY, sizeof(short*), (uint)pY);          Console.WriteLine(
"Address of pZ=&z is 0x{0:X}, size is {1}, value is 0x{2:X}",             (uint)&pZ, sizeof(double*),
(uint)pZ);          *pX = 20;          Console.WriteLine("After setting *pX, x = {0}", x);
Console.WriteLine("*pX = {0}", *pX);          pZ = (double*)pX;          Console.WriteLine("x treated as a
double = {0}", *pZ);          Console.ReadLine();       }    } }
```

This code declares four value variables:

- An int x
- A short y
- A byte y2
- A double z

It also declares pointers to three of these values: pX, pY, and pZ.

Next, you display the values of these variables as well as their sizes and addresses. Note that in taking the address of pX, pY, and pZ, you are effectively looking at a pointer *to* a pointer - an address of an address of a value. Notice that, in accordance with the usual practice when displaying addresses, you have used the {0:X} format specifier in the Console.WriteLine() commands to ensure that memory addresses are displayed in hexadecimal format.

Finally, you use the pointer pX to change the value of x to 20 and do some pointer casting to see what happens if you try to treat the content of x as if it were a double.

Compiling and running this code results in the following output. This screen output demonstrates the effects of attempting to compile both with and without the /unsafe flag:

```
 csc PointerPlayaround.cs Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42 for Microsoft (R)
Windows (R) 2005 Framework version 2.0.50727 Copyright (C) Microsoft Corporation 2001-2005. All rights
reserved. PointerPlayaround.cs(7,26): error CS0227: Unsafe code may only appear if         compiling with
/unsafe csc /unsafe PointerPlayaround.cs Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42 for
Microsoft (R) Windows (R) 2005 Framework version 2.0.50727 Copyright (C) Microsoft Corporation 2001-2005.
All rights reserved. PointerPlayaround Address of x is 0x12F4B0, size is 4, value is 10 Address of y is
0x12F4AC, size is 2, value is -1 Address of y2 is 0x12F4A8, size is 1, value is 4 Address of z is 0x12F4A0,
size is 8, value is 1.5 Address of pX=&x is 0x12F49C, size is 4, value is 0x12F4B0 Address of pY=&y is
0x12F498, size is 4, value is 0x12F4AC Address of pZ=&z is 0x12F494, size is 4, value is 0x12F4A0 After
setting *pX, x = 20 *pX = 20 x treated as a double = 2.86965129997082E-308
```

Checking through these results confirms the description of how the stack operates that was given in the “Memory Management under the Hood” section earlier in this chapter. It allocates successive variables moving downward in memory. Notice how it also confirms that blocks of memory on the stack are always allocated in multiples of 4 bytes. For example, y is a short (of size 2), and has the (decimal) address 1242284, indicating that the memory locations reserved for it are locations 1242284 through 1242287. If the .NET runtime had been strictly packing up variables next to each other, Y would have occupied just two locations, 1242284and 1242285.

The next example illustrates pointer arithmetic, as well as pointers to structs and class members. This example is called call PointerPlayaround2. To start off, you define a struct named CurrencyStruct, which represents a currency value as dollars and cents. You also define an equivalent class named CurrencyClass:

```
  struct CurrencyStruct {    public long Dollars;    public byte Cents;        public override string
ToString()    {       return "$" + Dollars + "." + Cents;    } } class CurrencyClass {    public long
Dollars;    public byte Cents;    public override string ToString()    {       return "$" + Dollars + "." +
Cents;    } }
```

Now that you have your struct and class defined, you can apply some pointers to them. Following is the code for the new example. Because the code is fairly long, we will g ze of CurrencyStruct, creating a couple

of CurrencyStruct instances and creating some CurrencyStruct pointers. You use the pAmount pointer to initialize the members of the amount1 CurrencyStruct and then display the addresses of your variables:

```
  public static unsafe void Main() {    Console.WriteLine(       "Size of CurrencyStruct struct is " +
sizeof(CurrencyStruct));    CurrencyStruct amount1, amount2;    CurrencyStruct* pAmount = &amount1;
long* pDollars = &(pAmount->Dollars);    byte* pCents = &(pAmount->Cents);    Console.WriteLine("Address of
amount1 is 0x{0:X}", (uint)&amount1);    Console.WriteLine("Address of amount2 is 0x{0:X}",
(uint)&amount2);    Console.WriteLine("Address of pAmount is 0x{0:X}", (uint)&pAmount);
Console.WriteLine("Address of pDollars is 0x{0:X}", (uint)&pDollars);    Console.WriteLine("Address of
pCents is 0x{0:X}", (uint)&pCents);    pAmount->Dollars = 20;    *pCents = 50;
Console.WriteLine("amount1 contains " + amount1);
```

Now you do some pointer manipulation that relies on your knowledge of how the stack works. Due to the order in which the variables were declared, you know that amount2 will be stored at an address immediately below amount1. The sizeof(CurrencyStruct) operator returns 16 (as demonstrated in the screen output coming up), so CurrencyStruct occupies a multiple of 4 bytes. Therefore, after you decrement your currency pointer, it will point to amount2:

```
  --pAmount;   // this should get it to point to amount2 Console.WriteLine("amount2 has address 0x{0:X} and
contains {1}",    (uint)pAmount, *pAmount);
```

Notice that when you call Console.WriteLine() you display the contents of amount2, but you haven't yet initialized it. What gets displayed will be random garbage - whatever happened to be stored at that location in memory before execution of the example. There is an important point here: normally, the C# compiler would prevent you from using an uninitialized variable, but when you start using pointers, it is very easy to circumvent many of the usual compilation checks. In this case, you have done so because the compiler has no way of knowing that you are actually displaying the contents of amount2. Only you know that, because your knowledge of the stack means that you can tell what the effect of decrementing pAmount will be. Once you start doing pointer arithmetic, you will find you can access all sorts of variables and memory locations that the compiler would usually stop you from accessing, hence the description of pointer arithmetic as unsafe.

Next, you do some pointer arithmetic on your pCents pointer. pCents currently points to amount1.Cents, but the aim here is to get it to point to amount2.Cents, again using pointer operations instead of directly telling the compiler that's what you want to do. To do this, you need to decrement the address pCents contains by sizeof(Currency):

```
  // do some clever casting to get pCents to point to cents // inside amount2 CurrencyStruct* pTempCurrency
= (CurrencyStruct*)pCents; pCents = (byte*) ( --pTempCurrency ); Console.WriteLine("Address of pCents is
now 0x{0:X}", (uint)&pCents);
```

Finally, you use the fixed keyword to create some pointers that point to the fields in a class instance and use these pointers to set the value of this instance. Notice that this is also the first time that you have been able to look at the address of an item stored on the heap rather than the stack:

```
  Console.WriteLine("\nNow with classes"); // now try it out with classes CurrencyClass amount3 = new
CurrencyClass(); fixed(long* pDollars2 = &(amount3.Dollars)) fixed(byte* pCents2 = &(amount3.Cents)) {
Console.WriteLine(       "amount3.Dollars has address 0x{0:X}", (uint)pDollars2);    Console.WriteLine(
"amount3.Cents has address 0x{0:X}", (uint) pCents2);    *pDollars2 = -100;    Console.WriteLine("amount3
contains " + amount3); }
```

Compiling and running this code gives output similar to this:

```
 csc /unsafe PointerPlayaround2.cs Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42 for
Microsoft (R) Windows (R) 2005 Framework version 2.0.50727 Copyright (C) Microsoft Corporation 2001-2005.
All rights reserved. PointerPlayaround2 Size of CurrencyStruct struct is 16 Address of amount1 is 0x12F4A4
Address of amount2 is 0x12F494 Address of pAmount is 0x12F490 Address of pDollars is 0x12F48C Address of
pCents is 0x12F488 amount1 contains $20.50 amount2 has address 0x12F494 and contains $0.0 Address of pCents
is now 0x12F488 Now with classes amount3.Dollars has address 0xA64414 amount3.Cents has address 0xA6441C
amount3 contains $-100.0
```

Notice in this output the uninitialized value of amount2 that is displayed and that the size of the CurrencyStruct struct is 16 - somewhat larger than you would expect given the sizes of its fields (a long and a byte should total 9 bytes). This is the effect of word alignment that was discussed earlier.

Using Pointers to Optimize Performance

Until now, all of the examples have been designed to demonstrate the various things that you can do with pointers. We have played around with memory in a way that is prob 's happening under the hood but

doesn't really help you to write better code. Here you're going to apply your understanding of pointers and see an example in which judicious use of pointers will have a significant performance benefit.

Creating Stack-Based Arrays

This section looks at one of the main areas in which pointers can be useful: creating high-performance, low-overhead arrays on the stack. As discussed in Chapter 2, "C# Basics," C# includes rich support for handling arrays. Although C# makes it very easy to use both 1-dimensional and rectangular or jagged multidimensional arrays, it suffers from the disadvantage that these arrays are actually objects; they are instances of System.Array. This means that the arrays are stored on the heap with all of the overhead that this involves. There may be occasions when you need to create a short-lived high-performance array and don't want the overhead of reference objects. You can do this using pointers, although as you see in this section, this is only easy for 1-dimensional arrays.

To create a high-performance array, you need to use a new keyword: stackalloc. The stackalloc command instructs the .NET runtime to allocate an amount of memory on the stack. When you call stackalloc, you need to supply it with two pieces of information:

- The type of data you want to store
- How many of these data items you need to store

For example, to allocate enough memory to store 10 decimal data items, you can write:

```
decimal* pDecimals = stackalloc decimal[10];
```

This command simply allocates the stack memory; it doesn't attempt to initialize the memory to any default value. This is fine for the purpose of this example because you are creating a high-performance array, and initializing values unnecessarily would hurt performance.

Similarly, to store 20 double data items, you write:

```
double* pDoubles = stackalloc double[20];
```

Although this line of code specifies the number of variables to store as a constant, this can equally be a quantity evaluated at runtime. So, you can write the previous example like this:

```
int size; size = 20;   // or some other value calculated at run-time double* pDoubles = stackalloc
double[size];
```

You will see from these code snippets that the syntax of stackalloc is slightly unusual. It is followed immediately by the name of the data type you want to store (and this must be a value type) and then by the number of items you need space for in square brackets. The number of bytes allocated will be this number multiplied by sizeof(*data type*). The use of square brackets in the preceding code sample suggests an array, which isn't too surprising. If you have allocated space for 20 doubles, then what you have is an array of 20 doubles. The simplest type of array that you can have is a block of memory that stores one element after another (see Figure 11-6).

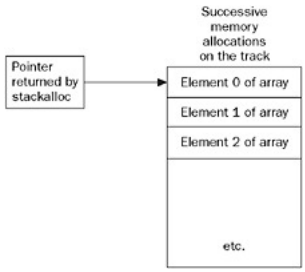

Figure 11-6

This diagram also shows the pointer returned by stackalloc, which is always a pointer to the allocated data type that points to the top of the newly allocated memory block. To use the memory block, you simply dereference the returned pointer. For example, to allocate space for 20 doubles and then set the first element (element 0 of the array) to the value 3.0, write this:

```
double* pDoubles = stackalloc double [20]; *pDoubles = 3.0;
```

To access the next element of the array, you use pointer arithmetic. As described earlier, if you add 1 to a pointer, its value will be increased by the size of whatever data type it points to. In this case, this will be just enough to take you to the next free memory location in the block that you have allocated. So, you can set the second element of the array (element number 1) to the value 8.4 like this:

```
double* pDoubles = stackalloc do                                    = 8.4;
```

By the same reasoning, you can access the element with index X of the array with the expression *(pDoubles+X).

Effectively, you have a means by which you can access elements of your array, but for general-purpose use this syntax is too complex. Fortunately, C# defines an alternative syntax using square brackets. C# gives a very precise meaning to square brackets when they are applied to pointers; if the variable p is any pointer type and X is an integer, then the expression p[X] is always interpreted by the compiler as meaning *(p+X). This is true for all pointers, not only those initialized using stackalloc. With this shorthand notation, you now have a very convenient syntax for accessing your array. In fact, it means that you have exactly the same syntax for accessing 1-dimensional stack-based arrays as you do for accessing heap-based arrays that are represented by the System.Array class:

```
 double* pDoubles = stackalloc double [20]; pDoubles[0] = 3.0;   // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4;   // pDoubles[1] is the same as *(pDoubles+1)
```

Tip This idea of applying array syntax to pointers isn't new. It has been a fundamental part of both the C and the C++ languages ever since those languages were invented. Indeed, C++ developers will recognize the stack-based arrays they can obtain using stackalloc as being essentially identical to classic stack-based C and C++ arrays. It is this syntax and the way it links pointers and arrays that was one of the reasons why the C language became popular in the 1970s, and the main reason why the use of pointers became such a popular programming technique in C and C++.

Although your high-performance array can be accessed in the same way as a normal C# array, a word of caution is in order. The following code in C# raises an exception:

```
  double[] myDoubleArray = new double [20]; myDoubleArray[50] = 3.0;
```

The exception occurs because you are trying to access an array using an index that is out of bounds; the index is 50, whereas the maximum allowed value is 19. However, if you declare the equivalent array using stackalloc, there is no object wrapped around the array that can perform bounds checking. Hence, the following code will *not* raise an exception:

```
  double* pDoubles = stackalloc double [20]; pDoubles[50] = 3.0;
```

In this code, you allocate enough memory to hold 20 doubles. Then you set sizeof(double) memory locations starting at the location given by the start of this memory + 50*sizeof(double) to hold the double value 3.0. Unfortunately, that memory location is way outside the area of memory that you have allocated for the doubles. There is no knowing what data might be stored at that address. At best, you may have used some currently unused memory, but it is equally possible that you may have just overwritten some locations in the stack that were being used to store other variables or even the return address from the method currently being executed. Once again, you see that the high performance to be gained from pointers comes at a cost; you need to be certain you know what you are doing, or you will get some very strange runtime bugs.

QuickArray Example

The discussion of pointers ends with a stackalloc example called QuickArray. In this example, the program simply asks users how many elements they want to be allocated for an array. The code then uses stackalloc to allocate an array of longs that size. The elements of this array are populated with the squares of the integers starting with 0 and the results displayed on the console:

```
  using System; namespace Wrox.ProCSharp.Memory {    class MainEntryPoint    {       static unsafe void
Main()       {          Console.Write("How big an array do you want? \n> ");          string userInput =
Console.ReadLine();          uint size = uint.Parse(userInput);          long* pArray = stackalloc long
[(int)size];          for (int i=0 ; i< size ; i++)          {             pArray[i] = i*i;          }
for (int i=0 ; i< size ; i++)          {             Console.WriteLine("Element {0} = {1}", i, *
(pArray+i));          }       }    } }
```

Here is the output for the QuickArray example:

```
 QuickArray How big an array do you want? > 15 Element 0 = 0 Element 1 = 1 Element 2 = 4 Element 3 = 9
Element 4 = 16 Element 5 = 25 Element 6 = 36 Element 7 = 49 Element 8 = 64 Element 9 = 81 Element 10 = 100
Element 11 = 121 Element 12 = 144 Element 13 = 169 Element 14 = 196
```

Professional C# 2005 with .NET 3.0

ISBN: 470124725 Year: 2007
EAN: N/A Pages: 427