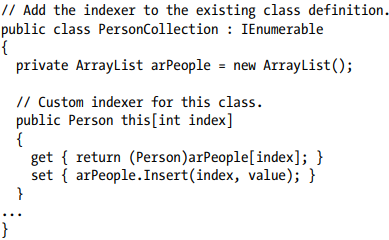
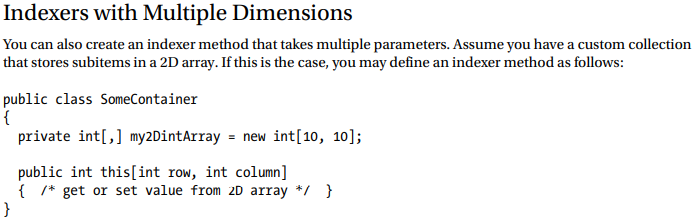
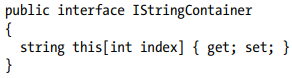
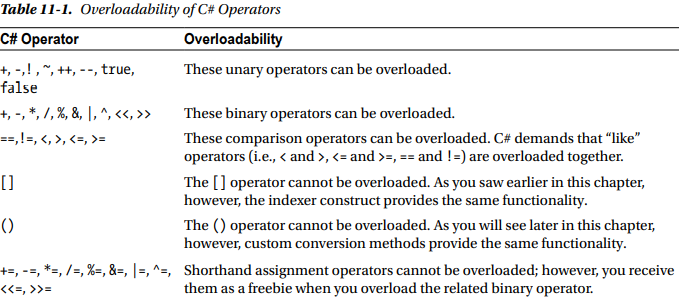
**Advanced C# Language Features**

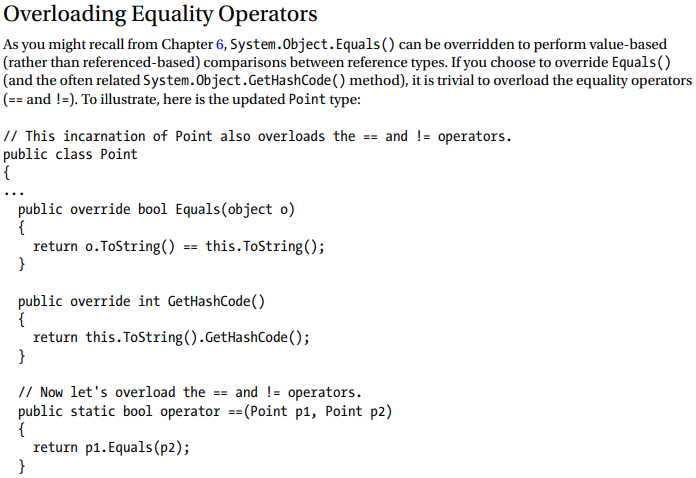
****To wrap things up, you’ll learn how to create an “unsafe” code context to directly manipulate unmanaged pointers. While it is certainly true that using pointers in C# applications is a fairly infrequent activity, understanding how to do so can be helpful in some circumstances that involve complex interoperability scenarios.

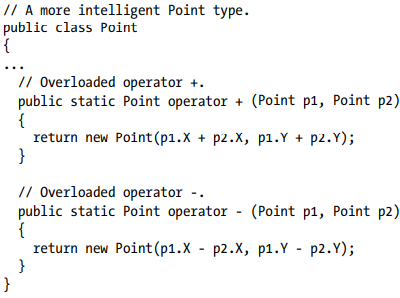
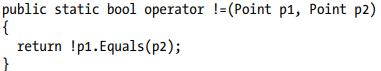
**Understanding Indexer Methods** As a programmer, you are certainly familiar with the process of accessing individual items contained within a simple array using the index operator ([]).

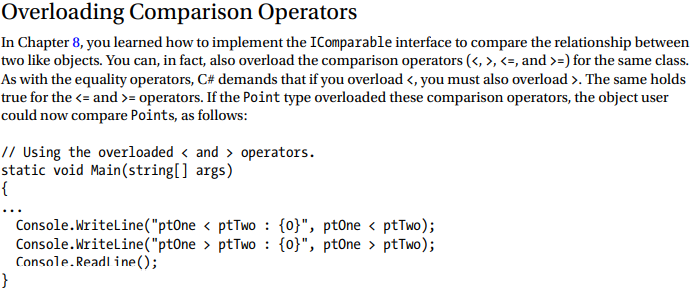


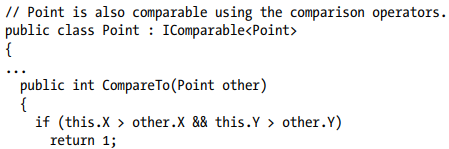
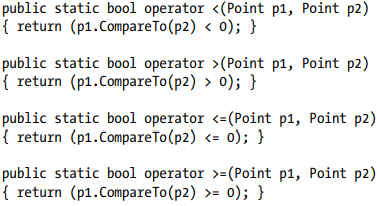


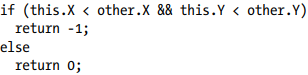
**Understanding Operator Overloading** C#, like any programming language, has a canned set of tokens that are used to perform basic operations on intrinsic types. For example, you know that the + operator can be applied to two integers to yield a larger integer.

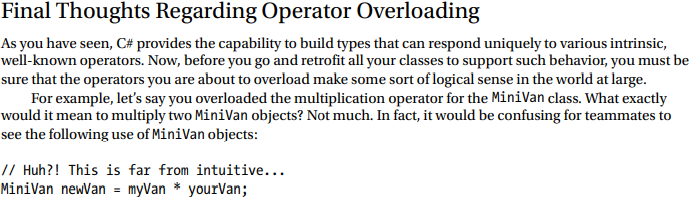


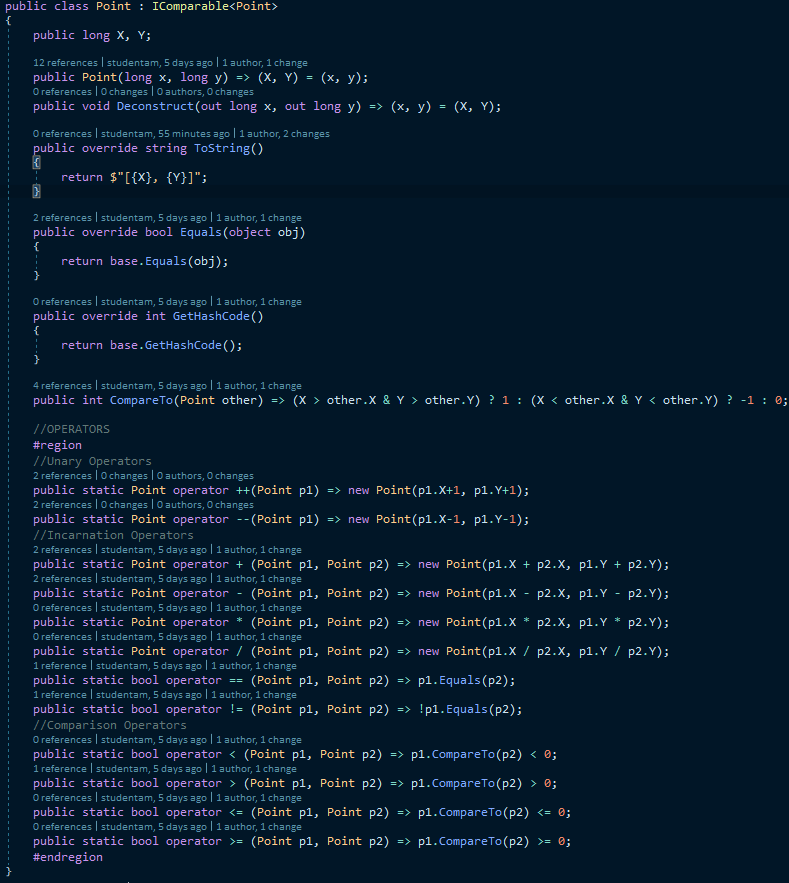




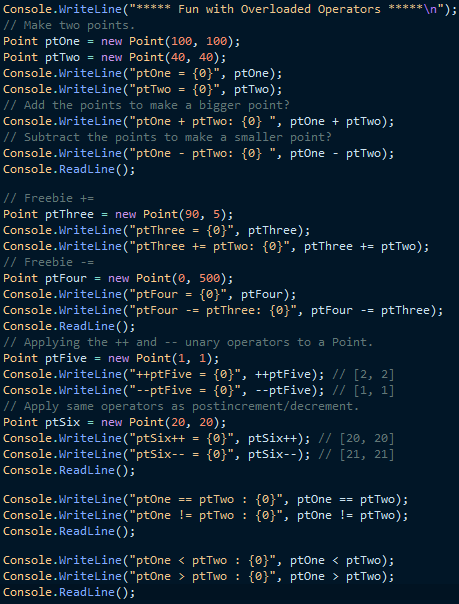


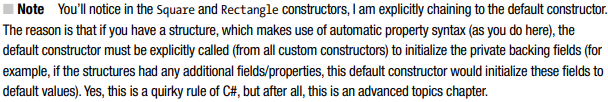






**Understanding Custom Type Conversions**

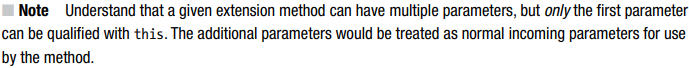
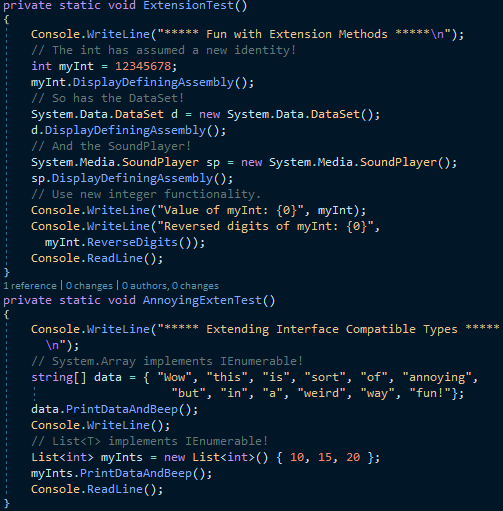
Let’s now examine a topic closely related to operator overloading: custom type conversions. To set the stage for the discussion, let’s quickly review the notion of explicit and implicit conversions between numerical data and related class types.

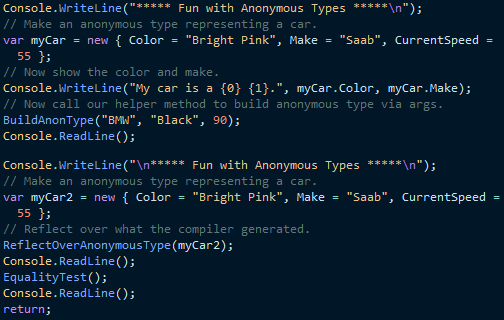
**Creating Custom Conversion Routines** Begin by creating a new Console Application project named CustomConversions. C# provides two keywords, explicit and implicit, that you can use to control how your types respond during an attempted conversion. Assume you have the following structure definitions:

**Additional Explicit Conversions for the Square Type** Now that you can explicitly convert Rectangles into Squares, let’s examine a few additional explicit conversions. Given that a square is symmetrical on all sides, it might be helpful to provide an explicit conversion routine that allows the caller to cast from an integer type into a Square (which, of course, will have a side length equal to the incoming integer). Likewise, what if you were to update Square such that the caller can cast from a Square into an int? Here is the calling logic:

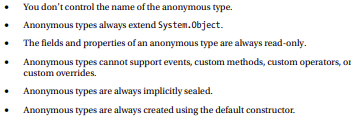
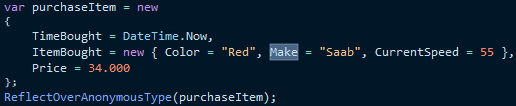
**Defining Implicit Conversion Routines** So far, you have created various custom explicit conversion operations. However, what about the following implicit conversion?

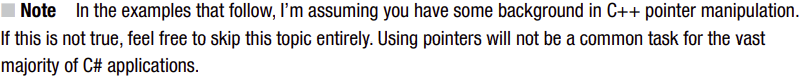
.Now here is the catch: it is illegal to define explicit and implicit conversion functions on the same type if they do not differ by their return type or parameter set.

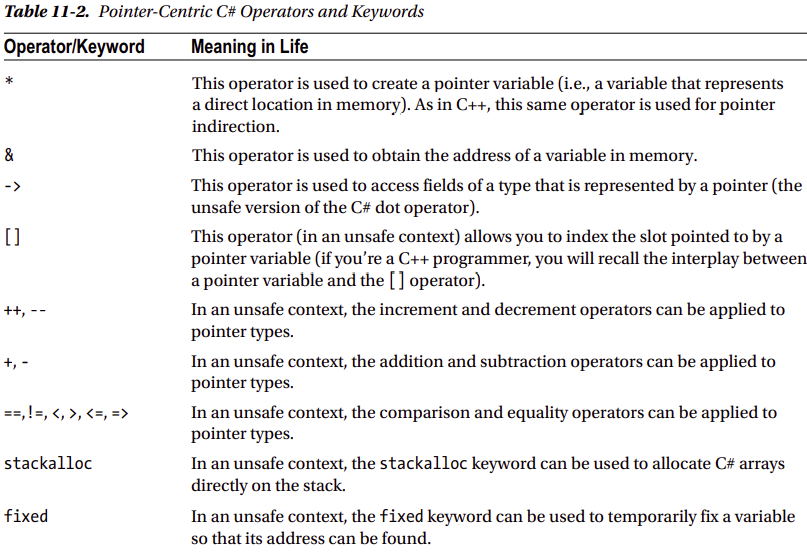
**Defining Extension Methods** When you define extension methods, the first restriction is that they must be defined within a static class (see Chapter 5) and, therefore, each extension method must be declared with the static keyword. The second point is that all extension methods are marked as such by using the this keyword as a modifier on the first (and only the first) parameter of the method in question. The “this qualified” parameter represents the item being extended.

**Understanding Anonymous Types** As an object-oriented programmer, you know the benefits of defining classes to represent the state and functionality of a given item you are attempting to model. To be sure, whenever you need to define a class that is intended to be reused across projects and that provides numerous bits of functionality through a set of methods, events, properties, and custom constructors, creating a new C# class is common practice.

**Anonymous Types Containing Anonymous Types**

It is possible to create an anonymous type that is composed of other anonymous types. For example, assume you want to model a purchase order that consists of a timestamp, a price point, and the automobile purchased. Here is a new (slightly more sophisticated) anonymous type representing such an entity:

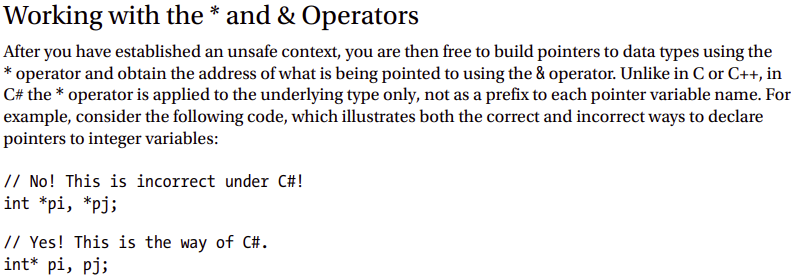
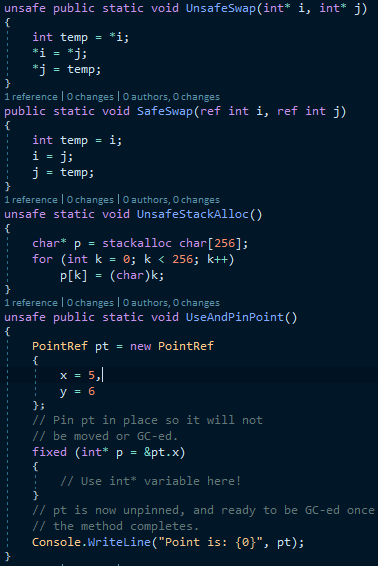
**Working with Pointer Types**

And now for the final topic of the chapter, which most likely will be the least used of all C# features for the vast majority of your .NET projects.

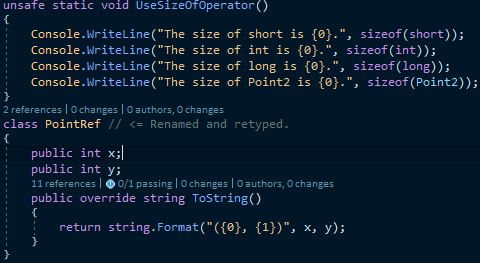
**An Unsafe (and Safe) Swap Function**

Of course, declaring pointers to local variables simply to assign their value (as in the previous example) is never required and not altogether useful. To illustrate a more practical example of unsafe code, assume you want to build a swap function using pointer arithmetic.

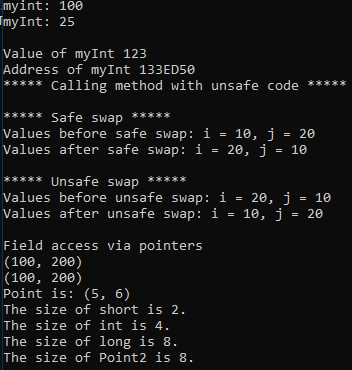
**Field Access via Pointers (the -> Operator)**

If you declare a pointer to a Point type, you will need to make use of the pointer field-access operator (represented by ->) to access its public members. As shown in Table 11-2, this is the unsafe version of the standard (safe) dot operator (.). In fact, using the pointer indirection operator (\*), it is possible to dereference a pointer to (once again) apply the dot operator notation. Check out the unsafe method:

**Pinning a Type via the fixed Keyword** As you saw in the previous example, allocating a chunk of memory within an unsafe context may be facilitated via the stackalloc keyword. By the very nature of this operation, the allocated memory is cleaned up as soon as the allocating method has returned (as the memory is acquired from the stack). However, assume a more complex example

**The stackalloc Keyword** In an unsafe context, you may need to declare a local variable that allocates memory directly from the call stack (and is, therefore, not subject to .NET garbage collection).

**The sizeof Keyword** The final unsafe-centric C# keyword to consider is sizeof. As in C++, the C# sizeof keyword is used to obtain the size in bytes of an intrinsic data type, but not a custom type, unless within an unsafe context.

**Summary** The purpose of this chapter was to deepen your understanding of the C# programming language. First, you investigated various advanced type construction techniques (indexer methods, overloaded operators, and custom conversion routines). Next, you examined the role of extension methods and anonymous types. As you’ll see in some detail in the next chapter, these features are useful when working with LINQ-centric APIs (though you can use them anywhere in your code, should they be useful). Recall that anonymous methods allow you to quickly model the “shape” of a type, while extension methods allow you to tack on new functionality to types, without the need to subclass. You spent the remainder of this chapter examining a small set of lesser-known keywords (sizeof, unsafe, and so forth) and during the process learned how to work with raw pointer types. As stated throughout the examination of pointer types, the vast majority of your C# applications will never need to use them.