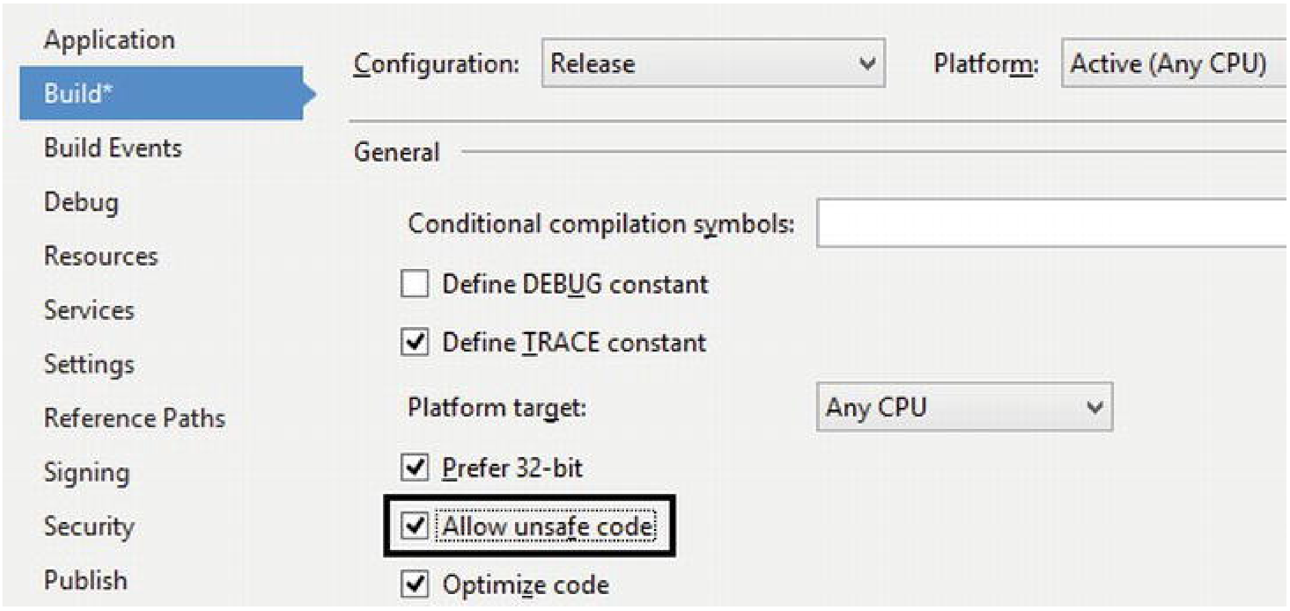
Unsafe Code

Managed code offers type safety, memory safety, and security guarantees, which eliminate some of the most difficult to diagnose bugs and security vulnerabilities prevalent in native code, such as heap corruptions and buffer overflows. This is made possible by prohibiting direct memory access with pointers, instead working with strongly-typed references, checking array access boundaries and ensuring only legal casting of objects.

However, in some cases these constraints may complicate otherwise simple tasks and reduce performance by forcing you to use a safe alternative. For example, one might read data from a file into a `byte[]` but would like to interpret this data as an array of `double` values. In C/C++, you would simply cast the `char` pointer to a `double` pointer. In contrast, in a safe .NET code, one could wrap the buffer with a `MemoryStream` object and use a `BinaryReader` object on top of the former to read each memory location as `double` values; another option is to use the `BitConverter` class. These solutions would work, but they're slower than doing this in unmanaged code. Fortunately, C# and the CLR support unsafe memory access with pointers and pointer casting. Other unsafe features are stack memory allocation and embedded arrays within structures. The downside of unsafe code is compromised safety, which can lead to memory corruptions and security vulnerabilities, so you should be very careful when writing unsafe code.

To use unsafe code, you must first enable compiling unsafe code in C# project settings (see Figure 8-1), which results in passing the `/unsafe` command-line parameter to the C# compiler. Next, you should mark the region where unsafe code or unsafe variables are permitted, which can be a whole class or struct, a whole method or just a region within a method.



Configuration: Release
Platform: Active (Any CPU)
Conditional compilation symbols:
☐ Define DEBUG constant
☑ Define TRACE constant
Platform target: Any CPU
☑ Prefer 32-bit
☑ Allow unsafe code
☑ Optimize code

Figure 8-1 . *Enabling unsafe code in C# project settings (Visual Studio 2012)*

Pinning and GC Handles

Because managed objects located on the GC heap can be relocated during garbage collections occurring at unpredictable times, you must pin them in order to obtain their address and prevent them from being moved around in memory.

Pinning can be done either by using the `fixed` scope (see example in Listing 8-1) in C# or by allocating a pinning GC handle (see Listing 8-2). P/Invoke stubs, which we will cover later, also pin objects in a way equivalent to the `fixed` statement. Use `fixed` if the pinning requirement can be confined to the scope of a function, as it is more efficient than the GC handle approach. Otherwise, use `GCHandle.Alloc` to allocate a pinning handle to pin an object indefinitely (until you explicitly free the GC handle by calling `GCHandle.Free`). Stack objects (value types) do not require pinning, because they're not subject to garbage collection. A pointer can be obtained directly for stack-located objects by using the ampersand (&) reference operator.

Listing 8-1. *Using fixed scope and pointer casting to reinterpret data in a buffer*

```
using (var fs = new FileStream(@"C:\Dev\samples.dat", FileMode.Open)) {
  var buffer = new byte[4096];
  int bytesRead = fs.Read(buffer, 0, buffer.Length);
  unsafe {
   double sum = 0.0;
   fixed (byte* pBuff = buffer) {
     double* pDblBuff = (double*)pBuff;
     for (int i = 0; i < bytesRead / sizeof(double); i++)
      sum + = pDblBuff[i];
      }
    }
  }
```

Caution The pointer obtained from the `fixed` statement must not be used outside the `fixed` scope because the pinned object becomes unpinned when the scope ends. You can use the `fixed` keyword on arrays of value types, on strings and on a specific value type field of a managed class. Be sure to specify structure memory layout.

GC handles are a way to refer to a managed object residing on the GC heap via an immutable pointer-sized handle value (even if the object's address changes), which can even be stored by native code. GC handles come in four varieties, specified by the `GCHandleType` enumeration: `Weak`, `WeakTrackRessurection`, `Normal` and `Pinned`. The `Normal` and `Pinned` types prevent an object from being garbage collected even if there is no other reference to it. The `Pinned` type additionally pins the object and enables its memory address to be obtained. `Weak` and `WeakTrackResurrection` do not prevent the object from being collected, but enable obtaining a normal (strong) reference if the object hasn't been garbage collected yet. It is used by the `WeakReference` type.

Listing 8-2. *Using a pinning GCHandle for pinning and pointer casting to reinterpret data in a buffer*

```
using (var fs = new FileStream(@"C:\Dev\samples.dat", FileMode.Open)) {
  var buffer = new byte[4096];
  int bytesRead = fs.Read(buffer, 0, buffer.Length);
  GCHandle gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);
  unsafe {
   double sum = 0.0;
   double* pDblBuff = (double *)(void *)gch.AddrOfPinnedObject();
   for (int i = 0; i < bytesRead / sizeof(double); i++)
     sum + = pDblBuff[i];
   gch.Free();
  }
}
```

Caution Pinning may cause managed heap fragmentation if a garbage collection is triggered (even by another concurrently running thread). Fragmentation wastes memory and reduces the efficiency of the garbage collector's algorithm. To minimize fragmentation, do not pin objects longer than necessary.

Lifetime Management

In many cases, native code continues to hold unmanaged resources across function calls, and will require an explicit call to free the resources. If that's the case, implement the `IDisposable` interface in the wrapping managed class, in addition to a finalizer. This will enable clients to deterministically dispose unmanaged resources, while the finalizer should be a last-resort safety net in case you forgot to dispose explicitly.

Allocating Unmanaged Memory

Managed objects taking more than 85,000 bytes (commonly, byte buffers and strings) are placed on the Large Object Heap (LOH), which is garbage collected together with Gen2 of the GC heap, which is quite expensive. The LOH also often becomes fragmented because it is never compacted; rather free spaces are re-used if possible. Both these issues increase memory usage and CPU usage by the garbage collector. Therefore, it is more efficient to use managed memory pooling or to allocate these buffers from unmanaged memory (e.g. by calling `Marshal.AllocHGlobal`). If you later need to access unmanaged buffers from managed code, use a "streaming" approach, where you copy small chunks of the unmanaged buffer to managed memory and work with one chunk at a time. You can use `System.UnmanagedMemoryStream` and `System.UnmanagedMemoryAccessor` to make the job easier.

Memory Pooling

If you are heavily using buffers to communicate with native code, you can either allocate them from GC heap or from an unmanaged heap. The former approach becomes inefficient for high allocation rates and if the buffers aren't very small. Managed buffers will need to be pinned, which promotes fragmentation. The latter approach is also problematic, because most managed code expects buffers to be managed byte arrays (`byte[]`) rather than pointers. You cannot convert a pointer to a managed array without copying, but this is bad for performance.

Tip You can look up the `% Time in GC` performance counter under the `.NET CLR Memory` performance counter category to get an estimate of the CPU time "wasted" by the GC, but this doesn't tell you what code is responsible. Use a profiler (see Chapter 2) before investing optimization effort, and see Chapter 4 for more tips on garbage collection performance.

We propose a solution (see Figure 8-2) which provides a copy-free access from both managed and unmanaged code and does not stress the GC. The idea is to allocate large managed memory buffers (segments) which reside on the Large Object Heap. There is no penalty associated with pinning these segments because they're already non-relocatable.

A simple allocator, where the allocation pointer (an index actually) of a segment moves only forward with each allocation will then allocate buffers of varying sizes (up to the segment size) and will return a wrapper object around those buffers. Once the pointer approaches the end and an allocation fails, a new segment is obtained from the segment pool and allocation is attempted again.

Segments have a reference count that is incremented for every allocation and decremented when the wrapper object is disposed of. Once its reference count reaches zero, it can be reset, by setting the pointer to zero and optionally zero filling memory, and then returned to the segment pool.

The wrapper object stores the segment's `byte[]`, the offset where data begins, its length, and an unmanaged pointer. In effect, the wrapper is a window into the segment's large buffer. It will also reference the segment in order to decrement the segment in-use count once the wrapper is disposed. The wrapper could provide convenience methods such a safe indexer access which accounts for the offset and verifies that access is within bounds.

Since .NET developers have a habit of assuming that buffer data always begins at index 0 and lasts the entire length of the array, you will need to modify code not to assume that, but to rely on additional offset and length parameters that will be passed along with the buffer. Most .NET BCL methods that work with buffers have overloads which take offset and length explicitly.

The major downside to this approach is the loss of automatic memory management. In order for segments to be recycled, you will have to explicitly dispose of the wrapper object. Implementing a finalizer isn't a good solution, because this will more than negate the performance benefits.

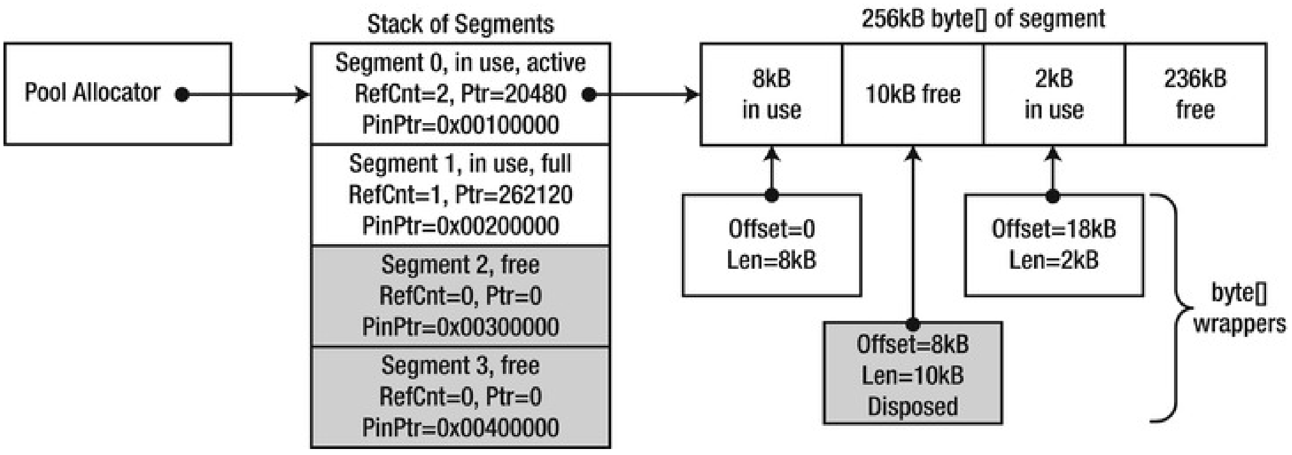


Figure 8-2. Proposed memory pooling scheme