Storage, Allocation, and Deallocation

Reference types are allocated exclusively from the managed heap, an area of memory managed by the .NET garbage collector, which will be discussed in more detail in Chapter 4. Allocating an object from the managed heap involves incrementing a pointer, which is a fairly cheap operation in terms of performance. On a multi-processor system, if multiple processors are accessing the same heap, some synchronization is required, but the allocation is still extremely cheap compared to allocators in unmanaged environments, such as `malloc`.

The garbage collector reclaims memory in a non-deterministic fashion and makes no promises regarding its internal operation. As we shall see in Chapter 4, a full garbage collection process is extremely expensive, but the average garbage collection cost of a well-behaved application should be considerably smaller than that of a similar unmanaged counterpart.

Note To be precise, there *is* an incarnation of reference types that can be allocated from the stack. Arrays of certain primitive types (such as arrays of integers) can be allocated from the stack using the `unsafe` context and the `stackalloc` keyword, or by embedding a fixed-size array into a custom struct, using the `fixed` keyword (discussed in Chapter 8). However, the objects created by the `stackalloc` and `fixed` keywords are not "real" arrays, and have a different memory layout than standard arrays allocated from the heap.

Stand-alone value types are usually allocated from the stack of the executing thread. However, value types can be embedded in reference types, in which case they are allocated on the heap, and can be boxed, transferring their storage to the heap (we will revisit boxing later in this chapter). Allocating a value type instance from the stack is a very cheap operation that involves modifying the stack pointer register (`ESP` on Intel x86), and has the additional advantage of allocating several objects at once. In fact, it is very common for a method's *prologue* code to use just a single CPU instruction to allocate stack storage for *all* the local variables present in its outermost block.

Reclaiming stack memory is very efficient as well, and requires a reverse modification of the stack pointer register. Due to the way methods are compiled to machine code, often enough the compiler is not required to keep track of the size of a method's local variables, and can destroy the entire stack frame in a standard set of three instructions, known as the function *epilogue*.

Below are the typical prologue and epilogue of a managed method compiled to 32-bit machine code (this is not actual production code produced by a JIT-compiler, which employs numerous optimizations discussed in Chapter 10). The method has four local variables, whose storage is allocated at once in the prologue and reclaimed at once in the epilogue:

```
int Calculation(int a, int b)
{
  int x = a + b;
  int y = a - b;
  int z = b - a;
  int w = 2 * b + 2 * a;
  return x + y + z + w;
}
; parameters are passed on the stack in [esp+4] and [esp+8]
push ebp
mov ebp, esp
add esp, 16 ; allocates storage for four local variablesmov eax, dword ptr [ebp+8]
add eax, dword ptr [ebp+12]
mov dword ptr [ebp-4], eax
; ...similar manipulations for y, z, w
mov eax, dword ptr [ebp-4]
add eax, dword ptr [ebp-8]
add eax, dword ptr [ebp-12]
add eax, dword ptr [ebp-16] ; eax contains the return value
mov esp, ebp ; restores the stack frame, thus reclaiming the local storage spacepop ebp
ret 8 ; reclaims the storage for the two parameters
```

Note The `new` keyword does not imply heap allocation in C# and other managed languages. You can allocate a value type on the stack using the `new` keyword as well. For example, the following line allocates a `DateTime` instance from the stack, initialized with the New Year's Eve (`System.DateTime` is a value type): `DateTime newYear = new DateTime(2011, 12, 31);`

What's the Difference Between Stack and Heap?

Contrary to popular belief, there isn't that much of a difference between stacks and heaps in a .NET process. Stacks and heaps are nothing more than ranges of addresses in virtual memory, and there is no inherent advantage in the range of addresses reserved to the stack of a particular thread compared to the range of addresses reserved for the managed heap. Accessing a memory location on

the heap is neither faster nor slower than accessing a memory location on the stack. There are several considerations that might, in certain cases, support the claim that memory access to stack locations is faster, overall, than memory access to heap locations. Among them:

- On the stack, temporal allocation locality (allocations made close together in time) implies spatial locality (storage that is close together in space). In turn, when temporal allocation locality implies temporal access locality (objects allocated together are accessed together), the sequential stack storage tends to perform better with respect to CPU caches and operating system paging systems.
- Memory density on the stack tends to be higher than on the heap because of the reference type overhead (discussed later in this chapter). Higher memory density often leads to better performance, e.g., because more objects fit in the CPU cache.
- Thread stacks tend to be fairly small – the default maximum stack size on Windows is 1MB, and most threads tend to actually use only a few stack pages. On modern systems, the stacks of all application threads can fit into the CPU cache, making typical stack object access extremely fast. (Entire heaps, on the other hand, rarely fit into CPU caches.)

With that said, you should not be moving all your allocations to the stack! Thread stacks on Windows are limited, and it is easy to exhaust the stack by applying injudicious recursion and large stack allocations.

Having examined the superficial differences between value types and reference types, it's time to turn to the underlying implementation details, which also explain the vast differences in memory density at which we hinted several times already. A small caveat before we begin: the details described below are internal implementation minutiae of the CLR, which are subject to change at any time without notice. We have done our best to ensure that this information is fresh and up to date with the .NET 4.5 release, but cannot guarantee that it will remain correct in the future.