Saving Memory with C# Structs

Posted on June 1, 2019

In C++, there's little difference between `class` and `struct`: a `class` defaults to "private" access and a `struct` defaults to "public" access. There are no other *technical* differences between the two, though structs are often used for small and simple data types.

C# mimics the syntax of C++ to some degree and also provides `class` and `struct`. However, in this case, the technical difference is quite large! In this post, I'll briefly explain that difference and highlight a scenario where using a `struct` saved a lot of memory on a project I worked on.

Types of Memory

Any time you create an object in code, the object takes up some amount of memory. The memory for the object must be allocated from some source of unused (or "free") memory. There are two possible sources for memory at runtime: the stack and the heap. Let's briefly examine the difference.

Stack

The stack is a contiguous area of memory that is allocated in a very simple and uniform fashion. Memory is allocated from the "bottom" of the stack (lower memory address) to the "top" of the stack (higher memory address). We can only deallocate the most recently allocated memory. If we wanted to deallocate the memory at the bottom of the stack, we'd first have to deallocate all the memory that had been allocated above it.

What memory is or isn't allocated on the stack is tracked with a simple memory pointer. When memory is allocated from the stack, the pointer is moved up by the appropriate amount. When memory is deallocated, the pointer moves back down. This makes allocation and deallocation on the stack extremely fast. When memory is deallocated, it isn't cleared - the pointer simply moves to a lower memory address and the deallocated memory will be overwritten the next time it is needed.

If you think about how a C++ or C# program works, it starts with a `main` function that calls other functions. Those functions call other functions, and so on. Eventually, each function returns to its caller. This structure works well for stack allocations.

When function A is called, local variables for function A exist on the stack. When function B is called, local variables for function B are allocated on the top of the stack. When function B returns to function A, function B's data is simply abandoned and we "pop" back to function A's data on the stack. A common mistake is to maintain a reference to B's memory after "popping" back to function A. The memory may still be valid for awhile...but it can be overwritten at any time that stack segment is reallocated!

Stack allocation is very efficient and works well for "function-local" data. If a piece of data will only ever be used in a single function, it can be declared locally in the function (and thus, allocated on the stack). Because of the way the stack works, it never becomes fragmented.

Heap

The stack works in a very simple and uniform way because it puts restrictions on how you can use that memory and how long objects stored in that memory will remain valid. Sometimes, those restrictions are unacceptable, and you need some other place to store data. That's where the heap comes in.

The heap is memory for arbitrary dynamic allocations. Unlike the stack, this is meant for allocating objects that will exist beyond the lifetime of a single function. Memory is allocated and deallocated from the heap in a potentially disorganized way, leading to the possibility of *fragmentation*. When memory is fragmented, allocations can fail if there's no contiguous chunk that's big enough for the allocation. For example, say total available memory is 1MB, but a small 32KB chunk is allocated right in the middle of this block; an allocation of 600KB would fail because not enough **contiguous** memory is free!

Whereas the stack essentially cleans up after itself, objects allocated on the heap need to be cleaned up when they are no longer needed. C++ leaves it to the programmer to handle this, but C# keeps track of this for you and runs a "garbage collector" on occasion to clean up allocated memory that's no longer being used.

Reference Types and Value Types

A `class` in C# is referred to as a *reference type*. This means that all class instances are allocated on the heap, and any variable of that type is a pointer to the object on the heap.

C# tries to do away with pointers for the sake of simplicity, but it actually uses pointers extensively - they're just hidden. Ironically, the desire to avoid pointers leaves us with a system where the vast majority of our variables act as pointers "under-the-hood"! Any variable that is a `class` type is really a pointer. This is why you must do null checks for most C# variables!

In addition to the object itself taking up memory, some additional memory overhead exists for `class` objects. Any variable for the object is really a pointer, so that pointer takes up 8 bytes in a 64-bit program.

Furthermore, some data (16 bytes in a 64-bit program) is stored per-object for internal C# purposes (such as garbage collection).

A `struct` in C# is referred to as a *value type*. Variables of this type are not pointers to objects - they ARE the objects! If you create a `struct` as a function-local variable, its memory will be allocated on the stack. If the `struct` instance is a class member variable, its memory will be allocated contiguously as part of the class instance's memory on the heap.

Structs in C# act a lot like a value type (or "non-pointer") in C++. When you perform an assignment operation, a copy is made. When you pass to a function, unless you pass by reference, a copy is made. Because assignment and passing to functions creates a copy, modifying the copy does not modify the original object. Struct variables do not need to be null checked. Interestingly, because C# wants to avoid dealing with pointers, it isn't possible to have a "pointer-to" a struct instance (beyond passing by reference to functions), which can be limiting in some cases.

The Ambiguity of the "New" Keyword

In C++, the `new` keyword is a dead giveaway that you are allocating memory from the heap. Value types are allocated without the `new` keyword.

```
void Example()
{
    MyClass* classPtr = new MyClass(); // allocated on the heap
    MyClass classValue; // allocated on the stack
}
```

This distinction is less clear in C#. In C#, both `struct` and `class` instances are created using the `new` keyword. So, the keyword really gives no indication as to whether we are allocating on the stack or heap.

The only way to know is to understand whether you are dealing with a `struct` or `class`.

```
void Example()
{
    Foo foo = new Foo(); // assuming Foo is a class, allocated on the heap
    Bar bar = new Bar(); // assuming Bar is a struct, allocated on the stack
}
```

There are a couple issues here. First, we only know whether our object is allocated on the heap or stack if we know whether it is a `struct` or `class`. In the above example, it's not super clear, unless you go look at the declaration of Foo and Bar. Second, C# removes your ability to choose heap or stack, which can be limiting: a simplification that also reduces flexibility.

In C#, classes are always allocated on the heap. Structs are allocated on the stack, if a local function variable, or on the heap as part of a class if a class member.

The Power of Structs

So, what's the point? Well, understanding this behavior in C# can improve the efficiency of your code. Let's look at a real world example where this made a huge difference.

For some time, Unity did not support serialization of `struct`, so all serializable types had to be classes. In that vein, for Skullgirls, we needed a small serializable class that is used for the game's internal scripting language:

```
[System.Serializable]
public class VarOrNum
{
    public float number;
    public byte flags;
    public byte index;
}
```

This class is used A LOT throughout the engine. When the game is running, there are usually about 2,000,000 instances in memory. They are uniquely referenced from other objects about 666,666 times (each has 3 instances).

Profiling this code sometime later, I noticed that we had unwittingly implemented a giant waste of memory - 15MB worth actually! And this memory could be easily reclaimed by making one small change:

```
[System.Serializable]
public struct VarOrNum
{
    public float number;
    public byte flags;
    public byte index;
}
```

The ONLY difference here is changing `class` to `struct`, or making this a *value type* instead of a *reference type*. Why does this save so much memory???

First, we need to realize that our 2,000,000 instances are all reference types when we use `class`, and we happen to know that each one is referenced a single time (aka one pointer per object). Our game is 64-bit, so the size of a pointer is 8 bytes. So, the amount of memory used just by pointers to SlotOrNum instances was about 15.26MB (8 * 2,000,000).

Second, how big do you think this class is? It consists of a float and two bytes, which adds up to 6 bytes. With padding for memory alignment, the size is 8 bytes. However, as described here (https://blogs.msdn.microsoft.com/seteplia/2017/05/26/managed-object-internals-part-1-layout/), each class instance in C# has management data associated with it; in a 64-bit application, this adds up to 16 bytes of additional overhead per object instance! So, the size of an instance is actually 16 + 8 = 24 bytes.

The amount of per-instance memory overhead is somewhat staggering at scale. The 16 byte header info plus the 8 byte pointer means that each instance has 24 bytes of overhead. Furthermore, you may notice that the variables I'm saving (float, byte, byte) can actually fit inside the 8 bytes of space being allocated for the pointer!

By switching to a `struct` we get two benefits that massively improve memory usage:

1. The 16 bytes of overhead that was tacked on to the object instance will no longer be allocated.
2. Instead of using 8 bytes for the pointer to the object instance in heap memory, the struct data is now just contiguously stored in memory - in other words, the float+byte+byte now occupy the memory that would have previously been used for an 8 byte pointer.

Also, since the struct version is allocated contiguously within class objects, there is one less "pointer dereference" when using these objects, which could improve memory access patterns and CPU caching. Following a pointer can lead to a cache miss, so fewer "pointer hops", the better.

Structs EVERYWHERE!?

Given these kinds of savings, are there any times we don't want to use structs? Well, unfortunately, structs have various limitations that can limit their usefulness.

C# value types are, by default, copied when you pass them as arguments or return them from functions. You can make use of the `ref` keyword to pass a value type by reference. There's no way to return a reference to a value type (which is great - that's a source of errors in C++). (EDIT: newer versions of C# do allow returning by reference, though the syntax is a bit unwieldy.)

Because of the copy creation, unless you use extreme care, you probably don't want to use `struct` for large objects that are passed around a lot without `ref` or returned from functions. Copying very large amounts of data can slow down your program.

The copy behavior can also be a massive source of errors - it's easy to accidentally make a copy of an object (especially if you are trying to refactor old code to convert a C# class to a struct). Accidental copies usually lead to bugs because you modify the copy and don't realize that the original object is no longer being modified.

Also, unlike C++, C# has no way to store a reference or pointer to a value type. As a result, if you want two variables that point to the same struct... you can't do it! The only thing you can do is store two variables that each have their own memory and contain the exact same values. This might be OK for smaller objects. But it can start to get complicated to think about, especially when the two variables should conceptually refer to one object. In these cases, a `class` may be better.

Conclusion

If you're coming from a C++ background, the choice of `struct` vs. `class` may be something you gloss over. But in C#, this seemingly simple choice can sometimes cost you a lot of memory, so choose wisely!

← PREVIOUS POST (HTTP://CLARKKROMENAKER.COM/POST/GENGINE-04-RENDERING/)

NEXT POST → (HTTP://CLARKKROMENAKER.COM/POST/GENGINE-05-ASSET-TYPES/)

Allen Warner • 9 months ago

First off, lot of great information here. It helped me better understand this process. But it seems like some pieces are still missing or I am not smart enough to follow. Here is what I see is going on: value types(including value type objects(struct)) are stored on the stack and reference type "objects" are stored on the heap. But the missing piece I see is, not only is the reference type object stored on the heap, a reference to an that object must also be stored on the stack. so even though the entire struct is stored on the stack, if it has members that are reference types then just the reference to those heap based objects will be stored on the stack instead of the entire object.

Clark Kromenaker Mod → Allen Warner • 9 months ago

Hi, thanks for the comment! I think you actually have a pretty good understanding based on your observation. Here's how I understand it at least:

1) If you create a value type in a function body, the memory for that is stored contiguously on the stack. If you create a reference type in a function body, an 8-byte pointer is stored on the stack, and that points to the object created on the heap.

2) If you allocate a struct on the stack, but the struct has reference type members (perhaps a List or other collection), then the struct simply stores an 8-byte pointer to the object on the heap. There is no way to avoid this, as far as I know!

3) If you allocate a class on the heap, but the class contains value type members, then those value type members exist contiguously in the heap memory that was allocated for that class. If the class has reference type members, 8-byte pointers to the heap memory for those members are stored.

If you have a ton of classes referencing other classes, you really end up with a ton of pointers to different parts of the heap at runtime. I think C# basically doesn't want you to think or worry about this, but then you try to make a mobile game with limited memory and you really gotta start thinking about it!

Honestly, I think this stuff is easier to understand with some visual examples...maybe I'll try to update the article. But I am no artist ;).

^ | v • Reply • Share ›

ReadyOK • a year ago

Hello great article! However, I'm still a bit confused. When you switched from class to struct in your own project, doesn't that mean the struct now became allocated in the heap as a member of the class? How then, is it possible to use the struct in such a way?

^ | v • Reply • Share ›

Clark Kromenaker Mod ➚ ReadyOK • a year ago

I also found this confusing at first. I'll try to explain a bit better, at least based on my understanding.

First, the easy part: if you create an instance of a class, it gets allocated on the heap. Any variable you have to that instance is basically a pointer.

For structs, the location they are allocated depends on where the struct lives. If you create a struct in a function, the struct lives on the stack. If a struct is a member variable in a class, then the struct exists *as part of the class instance* on the heap! If you have a class instance originally takes up 64 bytes on the heap, and then you add a 32 byte struct as a member variable, the class instance now takes up 96 bytes.

Remember, a struct can't be null. If you allocate an instance of a class that contains a struct member variable, enough space will be allocated for the members of the struct, with default values, even if you don't say "new MyStruct();" on the member variable.

I'm not sure if I understand your question "how is possible to use the struct in such a way?" but hopefully that explanation helps - let me know if I can clarify anything more.

^ | v • Reply • Share ›

ReadyOK ➚ Clark Kromenaker • a year ago

I think I got it. I was just unnecessarily confused. Thank you!

^ | v • Reply • Share ›

raephel • 2 years ago

Thanks for your article, it really helps a lot. However, I still have some questions.
That's say we have a Dictionary(for example: Dictionary<int,mydata> myData), and we need to get MyData by using a key, but it may be null if the key dosen't exist. Will it be better to use class to define MyData, or better to use Nullable struct?

^ | v • Reply • Share ›

Clark Kromenaker Mod ➚ raephel • 2 years ago

Hmm, good question. Storing structs in a Dictionary should be fine, though I think that you may not get as much of the benefit of contiguous storage of elements as you might get from a List or basic array.

When you use a class with Dictionary, there are three possibilities: the key doesn't exist in the dictionary, the key exists but has a null object, or the key exists and has a valid object.

When using a struct, there are only two possibilities: the key doesn't exist, or it does and points to a valid object. I'd argue that this is actually a pretty nice simplification.

But let's say you need to return a struct from a function, but you do the lookup in your Dictionary and find that no entry exists - what do you return? One option would be to return a "new/empty" instance of the struct, but it might be unclear to the caller whether the object is "valid" or not.

Another option is to have the function return bool (i.e. was an entry found?) and then use an "out" parameter to return the populated struct. For example, the Dictionary.TryGetValue method uses this approach. If you return false, the caller knows the item didn't exist in the Dictionary and assumes the passed in struct just contains "default" values.

^ | v • Reply • Share ›

Dennis19901 • 3 years ago

It's perfectly possible to avoid copying of value types (structs). It's also perfectly possible to "store" multiple references to a value type. Since C# 7.0, it's possible to return ref values and/or use ref locals. This is essentially the same as returning or using a pointer to the struct. This avoids copying and uses the actual reference to the struct in memory.

It's also possible to get pointers from structs as you would in C/C++. Allocate some memory the size of the struct (with Marshal), and store the struct in this memory. You can then freely use the pointer to this memory everywhere else to reference to this memory.

^ | v • Reply • Share ›

Clark Kromenaker Mod ➚ Dennis19901 • 3 years ago

Hi, thanks for clarifying! Since I do most of my C# coding in Unity, I haven't always been up to speed on the latest and greatest C# features. I think Unity only fairly recently (in the last year or so) updated their C# version to 6.0. When Unity officially uses 7.0, I'll look into using those features.

I've used Marshaling a bit with some C++ native plugins for our games. It's a good point that we can get pointers to structs (and use pointers in general in C#) if we are determined enough.

^ | v • Reply • Share ›

Dennis19901 ➚ Clark Kromenaker • 3 years ago

Returning references and local references are available in Unity as we speak.

(mailto:kromenak@gmail.com) (https://github.com/kromenak)

(https://twitter.com/kromenak) (https://linkedin.com/in/clarkkromenaker)

(https://www.instagram.com/kromenak) (http://clarkkromenaker.com/index.xml)