# Почему нельзя наследоваться от value type

Проблема вот в чём. Допустим, разрешено было бы наследоваться от структур.

Рассмотрим такой код:

class C1

{

int x;

}

class C2 : C1

{

int y;

public override string ToString() { return y.ToString(); }

}

Вы можете написать так: C1 c = new C2(); и вызвать c.ToString();. Поскольку c является лишь ссылкой на данные, проблем с вызовом не возникает. Это стандартная фича, базовое полиморфное поведение ссылочных типов.

Представьте себе теперь аналогичную ситуацию:

struct S1

{

int x;

}

struct S2 : S1

{

int y;

public override string ToString() { return y.ToString(); }

}

S1 s = new S2();

Что будет в s? Из соображений эффективности структуры хранятся в памяти *как есть*, то есть, просто набор полей, а не ссылка на них. Это значит, что под s отводится столько байт, сколько занимает S1, и для поля y там просто нет места! Что в этом случае должен вернуть вызов s.ToString()?

Аналогичная проблема возникает при передаче параметров производного типа в функцию, требующую аргумент базового типа.

Эта проблема называется ***slicing***, и она актуальна для дизайна многих языков программирования. Например, она присутствует в C++. Решение в C# — отказаться от наследования структур: допустить неработающий полиморфизм при наследовании — плохая идея. При таком решении slicing в C# исключён.

Другие языки решают проблему по-другому. Например, в C++ настоящим типом объекта s будет просто S1, а не S2, и y просто потеряется. Создатели языка C# считают такое поведение неинтуитивным, ведущим к проблемам и ошибкам, поэтому они пошли по другому пути. Если вам нужно наследование, в C# нужно воспользоваться классами, или перейти к композиции.

Проблему *можно* было бы решить, если бы структуры хранились точно так же, как и классы: не значение, а ссылка на него. При этом операции наподобие присвоения должны были бы для сохранения семантики копировать не ссылку, а *значение* всё равно. Но такой подход, хоть и разрешил бы наследование структур, сильно ухудшил бы их *эффективность*: каждая операция со структурой стала бы медленнее из-за лишнего косвенного доступа, плюс аллокация структур в общем случае была бы вытеснена из стека в кучу.

Архитекторы языка разменяли полиморфизм структур на их эффективность.

# Почему быстрее создать лист классов, чем лист интов?

the reasons are probably mostly due to the list needing to allocate at least two objects, possibly more depending on how it is optimized.

For high performance code a common guideline is to avoid high frequency allocations. While allocations in c# are fast, they still take some time to manage. This often means sticking with fixed size arrays, or at least set the capacity of any lists on creation.

Another important point is using structs, they are stored directly in the list/array, instead of storing a reference to a separate object. This avoids some [object overhead](https://stackoverflow.com/questions/10655829/what-is-the-memory-overhead-of-a-net-object), removes the memory need of a separate reference, and ensures all values are stored sequentially in memory. All of this help ensure caches are used efficiently. Using a smaller datatype like short/ushort may also help if that is possible. Note that structs should preferably be [immutable](https://stackoverflow.com/questions/441309/why-are-mutable-structs-evil), and there are some keywords like 'ref' and ['in'](https://devblogs.microsoft.com/premier-developer/the-in-modifier-and-the-readonly-structs-in-c/) that can help avoid the overhead of copying data.

In some specific cases it can be an idea to separate values into different arrays, i.e. one for all the FromSquare values, one for all the ToSquare values etc. This can be a benefit if an operation mostly uses only a single value, again benefiting from better cache usage. It might also make SIMD easier to apply, but that might not apply in this case.

Moreover, when when measuring performance, at least use a [stopwatch](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0). That is much more accurate than dateTime, and no harder to use. Benchmark.Net would be even better since that helps compensate for various sources of noise, and can benchmark for multiple platforms. A good profiler can also be useful since it can give hints at what takes most time, how much you are allocating etc.

# Как уменьшить количество аллокаций массивов (ArrayPool)

Remember, using ArrayPool instead of allocating with new puts the responsibility of freeing the memory on you. Your application will not leak memory if you don't guarantee that the Return method is called, but the ArrayPool is prevented from reusing the memory, thus denying the benefits you gain from using ArrayPool.

In simple use cases where you create a buffer and release it in the same method, it makes sense to put it into a finally clause:

private static void LocalUseOfSharedPool(int i)

{

int[] arr = ArrayPool<int>.Shared.Rent(ARRAYSIZE);

try

{

ShowAddress($"simple array {i}", arr);

FillTheArray(arr);

UseTheArray(arr);

}

finally

{

ArrayPool<int>.Shared.Return(arr);

}

}

In more complex cases you must make sure to not leak the memory in other ways.

Also note, that your buffer now has a lifetime. If you pass your array to another object B , you need to make sure, that object B is not using the array after your call to ArrayPool<>.Return.

Since using ArrayPool is a performance issue, measure the gains of using it, especially if you want to change an existing system.

# Когда не сработает finally?

There are a number of inaccuracies in the other answers.

Control is passed to the finally block when control leaves the try block normally -- that is, by a return, goto, break, continue, or simply falling off the end. Control is passed to the finally block when control leaves the try block via an exception that has been caught by an enclosing catch block.

In every other circumstance there is no guarantee that the code in the finally block will be called. In particular:

* If the try block code goes into an infinite loop, or the thread is frozen and never unfrozen, then the finally block code is never called.
* If the process is paused in the debugger and then aggressively killed then the finally block is never called. If the process does a fail-fast then the finally block is never called.
* If the power cord is pulled out of the wall then the finally block is never called.
* If there is an exception thrown without a corresponding catch block then whether the finally block runs or not is an implementation detail of the runtime. The runtime can choose any behaviour when there is an uncaught exception. Both "do not run the finally blocks" and "do run the finally blocks" are examples of "any behaviour", so either can be chosen. Typically what the runtime does is ask the user if they want to attach a debugger before the finally blocks run; if the user says no then the finally blocks run. But again: the runtime is not required to do that. It could just fail fast.

You cannot rely on finally blocks always being called. If you require a strong guarantee about code executing then you should not be writing a try-finally, you should be writing a [constrained execution region](https://learn.microsoft.com/en-us/dotnet/framework/performance/constrained-execution-regions). Writing a CER correctly is one of the most difficult tasks in C# programming, so study the documentation carefully before you try to write the code.

Incidentally, a "fun fact" about finally-blocked gotos is:

try { goto X; } finally { throw y; }

X : Console.WriteLine("X");

X is an unreachable label targetted by a reachable goto! So next time you're at a party you can be like "hey everybody, can anyone make a C# program that has an unreachable label that is targetted by a reachable goto?" and you'll see who at the party has read the reachability specification and who has not!