Handles are the better pointers

Jun 17, 2018

28-Nov-2018: I've added a small update at the end on how to prevent 'handle collisions' with a per-slot generation counter

Original Post:

…wherein I talk a bit about how I'm doing dynamic memory management in C and C++ these days which basically replaces raw- and smart-pointers with 'index-handles'.

In my last blog post I was mentioning pointer- and allocation-free programming, but was skipping over the details. This is what the following blog post is about.

This is all based on the (sometimes painful) experience of wrestling for 15+ years with fairly big C++ code bases (0.5 to around 1 mloc) where memory is often managed through smart pointers. The worst case being tens- to hundreds-of-thousands of small C++ objects, each in its own heap allocation, pointing to each other through smart pointers. While such code is quite robust in terms of memory corruption (segfaults and corruption rarely happens, since most attempts are caught by asserts when dereferencing smart pointers), this type of 'object spiderweb code' is also dog-slow without obvious starting points for optimization, since the entire code is full of cache misses. Other typical problems are memory fragmentation and 'fake memory leaks' because a forgotten smart pointer prevents freeing the underlying memory (I call them 'fake leaks' because this type of leaks cannot be caught by memory debugging tools).

Nothing presented here is particularly new or clever, it's just a collection of a few simple ideas which together work fairly well in bigger code bases to prevent (or at least detect early) a number of common memory-related problems in C and C++, and may even be useful in higher-level garbage-collected languages to reduce pressure on the garbage collector.

However, the underlying design philosophy doesn't fit very well into a classical OOP world where applications are built from small autonomous objects interacting with each other. That's why it's also quite tricky to implement those ideas in a big existing OOP code base, where object creation and destruction happens 'decentralized' all over the code.

The approach described here works very well though with a data-oriented architecture, where central systems work on arrays of data items packed tightly in memory.

Most of the following blog post is written from a game developer's perspective, but should also apply to other areas where a program needs to juggle a few hundred to a few million objects (or generally 'data items') in memory, and where such items are created and destroyed frequently.

The gist is:

- move all memory management into centralized systems (like rendering, physics, animation, …), with the systems being the sole owner of their memory allocations
- group items of the same type into arrays, and treat the array base pointer as system-private
- when creating an item, only return an 'index-handle' to the outside world, not a pointer to the item
- in the index-handles, only use as many bits as needed for the array index, and use the remaining bits for additional memory safety checks
- only convert a handle to a pointer when absolutely needed, and don't store the pointer anywhere

I'll explain each of those points in detail below. But the idea is basically that common 'user-level' code doesn't directly call memory allocation functions (like malloc, new or make_shared/make_unique), and reduces the use of pointers to an absolute minimum (only as short-lived references when direct memory access to an item is absolutely needed). Most importantly, pointers are never the 'owner' of an item's underlying memory.

Instead, direct memory manipulation happens as much as possible inside a few centralized systems where memory-related problems are easier to debug and optimize.

Move memory management into central systems

In this blog post, a ‘system’ is a (usually fairly big) part of a code base which takes care of a number of related tasks, like ‘rendering’, ‘physics’, ‘AI’, ‘character animation’ and so on. Such a system is separated from other systems and ‘user-code’ through a clearly defined function API, and the work that happens on the system’s data is performed in tight central loops instead of being spread out all over the code base.

Systems often work on items created and destroyed under control of user code (but note that creation and destruction of items is different from allocating and freeing the memory used by those items!). For instance a rendering system might deal with vertex buffers, textures, shaders and pipeline state objects. A physics system works with rigid bodies, joints and collision primitives, and an animation system works with animation keys and curves.

It makes sense to move the memory management for such items into the systems themselves, because a general memory allocator doesn’t have the system-specific ‘domain knowledge’ about how data items are processed and the relationships between data items. This allows the system to optimize memory allocations, perform additional validation checks when creating and destroying items, and arrange items in memory for making best use of the CPU’s data caches.

A good example for this ‘system domain knowledge’ is the destruction of rendering resource objects with modern 3D APIs: a resource object can not simply be destroyed immediately when the user code says so because the resource might still be referenced in a command list waiting to be consumed by the GPU. Instead the rendering system would only mark the resource object for destruction when the user code requests its destruction, but the actual destruction happens at a later time when the GPU no longer uses the resource.

Group items of the same type into arrays

Once all memory management has moved into systems, the system can optimize memory allocations and memory layout with its additional knowledge about how items are used. One obvious optimization is to reduce the number of general memory allocations by grouping items of the same type into arrays, and allocate those arrays at system startup.

Instead of performing a memory allocation each time a new item is created, the system keeps track of free array slots, and picks the next free slot. When the user code no longer needs the item, the system simply marks the slot as free again instead of performing a deallocation (not different from a typical pool allocator).

This pool allocation is most likely a little bit faster than performing a memory allocation per item, but this is not even the main reason for keeping items in arrays (modern general allocators are quite fast too for small allocations).

A few additional advantages are:

- items are guaranteed to be packed tightly in memory, general allocators sometimes need to keep some housekeeping data next to the actual item memory
- it’s easier to keep ‘hot items’ in continuous memory ranges, so that the CPU can make better use of its data caches
- it’s also possible to split the data of a single item into several subitems in different arrays for even tighter packing and better data cache usage (AoS vs SoA and everything inbetween), and all those data layout details remain private to the system and are trivial to change without affecting ‘outside code’
- as long as the system doesn’t need to reallocate arrays, it is guaranteed that there will be no memory fragmentation (although this is less of an issue in a 64-bit address space)
- it’s easier to detect memory leaks early, and provide more useful error messages: when a new item is created a system can trivially check the current number of items against an expected upper bound (for instance a game might know that there should never be more

than 1024 textures alive at a time, and since all textures are created through the rendering system, the system can print out a more useful warning message when this number is exceeded)

Public index-handles instead of pointers

Keeping system items in arrays instead of unique allocations has the advantage that an item can be identified through an array index instead of requiring a full pointer. This is very useful for memory safety. Instead of handing memory pointers to the outside world, the system can treat the array base pointers as 'private knowledge', and only hand out array indices to the public. Without the base pointer to compute an item's memory location, the outside code can't access the item's memory, even with a lot of criminal energy.

This has a number of further advantages:

- In many situations, code outside the system never even needs to directly access memory of an item, only the system does. In such an 'ideal' situation, user code never accesses memory through pointers, and can never cause memory corruption.
- Since only the system knows the array base pointers, it's free to move or reallocate the item arrays at will without invalidating existing index handles.
- Array indices need fewer bits than full pointers, and a smaller data type can be picked for them, which in turn allows tighter packing of data structures and better data cache usage (this has the caveat that additional handle bits can be used to increase memory safety, more about this below)

If user code needs to access the memory of an item directly it needs to obtain a pointer through a 'lookup function' which takes a handle as input and returns a pointer. As soon as such a lookup function exists, the fairly watertight memory safety scenario outlined above is no longer guaranteed, and the user code should adhere to a few rules:

- pointers should never be stored anywhere, since the next time the pointer is used it may no longer point to the same item, or even to valid memory

- a pointer should only be used in a simple code block, and not ‘across’ function calls

Every time a handle is converted into a pointer, the system can guarantee that the returned pointer still points to the same item that the handle was originally created for (more on this below), but this guarantee ‘decays’ over time since the item under the pointer may have been destroyed, or the underlying memory may have been reallocated to a different location (this is the same problem as iterator invalidation in C++).

The two simple rules above are easy to memorize and are a good compromise between not exposing pointers to user code at all, and having a (somewhat costly) handle-to-pointer conversion for every single memory access.

Memory safety considerations

First, each type of handle should get its own C/C++ type, so that attempting to pass the wrong handle type to a function is already detected at compile time (note that a simple typedef isn’t enough to produce a compiler warning, the handle must be wrapped into its own struct or class - however this could be limited to debug compilation mode).

All runtime memory safety checks happen in the function which converts a handle into a pointer. If the handle is just an array index, it would look like this:

- a range check happens for the index against the current item array size, this prevents segmentation faults and reading or writing allocated but unrelated memory areas
- a check needs to happen whether the indexed array item slot contains an active item (is not currently a ‘free slot’), this prevents the simple variant of ‘use after free’
- finally the item pointer is computed from the private array base pointer and public item index

The resulting pointer is safe to use as long as:

- the item array isn’t reallocated
- the indexed item hasn’t been destroyed

Both can only happen inside one of the system's functions, that's why the two 'pointer usage rules' exist (don't store pointers, don't keep pointers across function calls).

There is a pretty big hole in the above use-after-free check though: if we only check if an array slot behind an index-handle contains a valid item, it's not guaranteed that it is the *same* item the handle was originally created for. It can happen that the original item was destroyed, and the same array slot was reused for a *new* item.

This is where the 'free bits' in a handle come in: Let's say our handles are 16-bits, but we only ever need 1024 items alive at the same time. Only 10 index bits are needed to address 1024 items, which leaves 6 bits free for something else.

If those 6 bits contain some sort of 'unique pattern', it's possible to detect dangling accesses:

- When an item is created, a free array item is picked and its index is put into the lower 10 handle bits. The upper 6 bits are set to a 'unique bit pattern'
- The resulting 16-bit handle (10 bits index + 6 bit 'unique pattern') are returned to the outside world, and at the same time stored with the array slot.
- When an item is destroyed, the item handle stored with the array slot is set to the 'invalid handle' value (can be zero, as long as zero is never returned as a valid handle to the outside world)
- When the handle is converted to a pointer, the lower 10 bits are used as array index to lookup the array slot, and the entire 16-bit handle is compared against the handle that's currently stored with the array slot:
  - if both handles are equal, the pointer is valid and points to the same item the handle was created for
  - otherwise this is a dangling access, the slot item has either been destroyed (in that case the stored handle would have the 'invalid handle' value), or has been destroyed and reused for a new item (in that case the upper 6 'unique pattern' bits don't match)

This handle-comparison check when converting a handle to a pointer works quite well to detect dangling-accesses, but it isn't waterproof because the same combination of array index and 'unique pattern' will be created sooner or later. But it’s still better than no dangling protection at all (like raw pointers), or a ‘fake memory leak’ which would happen in similar situations with smart pointers.

Finding good strategies to create unique handles that collide as rarely as possible is the most important part of course, and left as an excercise to the reader ;P

Obviously it’s good to use as many bits for the unique pattern as possible, and the way how free array slots are reused is important as well (e.g. LIFO vs FIFO). It’s probably also good to write a little creation/destruction stress-test which checks for handle collisions and can be used to tweak the unique-pattern creation for specific use cases. A system where items are created and destroyed with a very high frequency needs more effort (or simply more handle bits) than systems where item creation and destruction is a rare occurance.

Other useful properties of handles

Apart from the whole memory safety aspect, handles are also useful for other situations where pointers are problematic:

Handles can be used as shared object identifiers across processes (all you need is some sort of ‘create_item_with_handle()’ function, which doesn’t create a new handle, but takes an existing handle as input argument). This is especially useful for online games where handles can be shared between the server and all clients in a game session, or in savegame systems to store references to other objects.

Sometimes it’s useful to create a whole group of related items (for instance animation keys and curves), and reference the whole item group with a single handle. In that case some sort of ‘range handle’ can be used, which contains not only an index (of the first item), but also the number of items in the range.

In some situations it’s also useful to reserve a few handle bits for an item type if static type checking at compile time isn’t sufficient.

In conclusion, I find it quite surprising how naturally and elegantly handles solve many problems I encountered in the past with the traditional ‘pointers to objects on the heap’ model, and how little I miss this model now (and the parts of C++ that are built around it).

Some real-world examples

The sokol-gfx API is an example of a C-API which uses handles instead of pointer of rendering resource objects (buffers, images, shaders, …):

sokol_gfx.h

The Oryol Gfx module is a similar 3D API wrapper, but written in C++:

Oryol Gfx Module

The Oryol Animation extension module is a character animation system which keeps all its data in arrays:

Oryol Anim Module

Update 28-Nov-2018

…in the post above I was sort of handwaving away the problem of creating the same unique-tag twice for the same slot, and a nice person on twitter hinted me about a very simple, elegant and embarrassingly 'obvious' solution:

Each array slot gets its own **generation counter**, which is bumped when a handle is released (can also happen when the handle is created, but bumping on release means you don't need a reserved value for "free slots" to detect an invalid handle).

To check if a handle is valid, simply compare its unique-tag with the current generation counter in its slot.

Once the generation counter would 'overflow', **disable** that array slot, so that no new handles are returned for this slot.

This is a perfect solution for avoiding handle collisions, but the handles will eventually run out since all array slots will be disabled eventually. But since each slot has its own counter, this only happens after *all* handle-bits are exhausted, not just the few unique-tag bits.

So with 32-bit handles, you can always create 4 billion items, with at most 2^(32 - num_counter_bits) alive at the same time. This also means the number of bits for the unique-tag can be reduced without compromising ‘handle safety’.

It may also be possible to re-activate disabled slots once it can be guaranteed that no more handles for that slot are out in the wild (maybe at special places in the code like entering or exiting a level).

The Brain Dump

The Brain Dump
floooh@gmail.com

floooh
flohofwoe

This is the blog and personal web page of Andre Weissflog (Floh, floooh, flohofwoe) mostly about programming stuff.