

Scope Stack Allocation

Andreas Fredriksson, DICE
<dep@dice.se>

Contents

- What are Scope Stacks?
- Background – embedded systems
- Linear memory allocation
- Scope Stacks
- Bits and pieces

What are Scope Stacks?

- A memory management tool
 - Linear memory layout of arbitrary object hierarchies
- Support C++ object life cycle if desired
 - Destructors called in correct dependency order
- Super-duper oiled-up fast!
- Makes debugging easier

Background

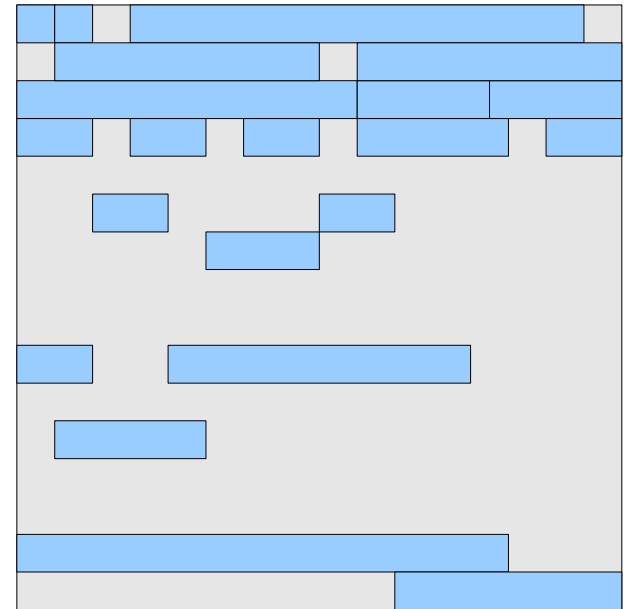
- Why is all this relevant?
- Console games are embedded systems
 - Fixed (small) amount of memory
 - Can't run out of memory or you can't ship – fragmentation is a serious issue
- Heap allocation is very expensive
 - Lots of code for complex heap manager
 - Bad cache locality
 - Allocation & deallocation speed

Embedded systems

- With a global heap, memory fragments
 - Assume 10-15% wasted with good allocator
 - Can easily get 25% wasted with poor allocator
- Caused by allocating objects with mixed life time next to each other
 - Temporary stuff allocated next to semi-permanent stuff (system arrays etc)

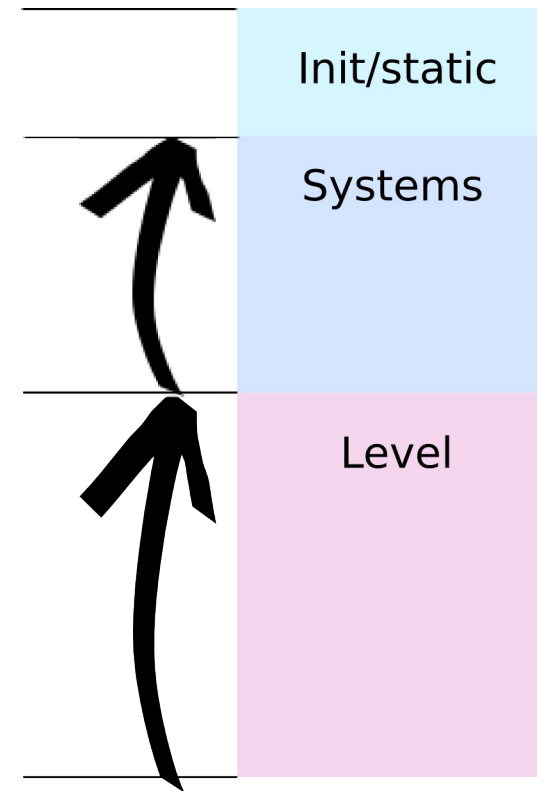
Heap Fragmentation

- Each alloc has to traverse the free list of this structure!
 - Assuming “best fit” allocator for less fragmentation
 - Will likely cache miss for each probed location
 - Large blocks disappear quickly



Memory Map

- Ideally we would like fully deterministic memory map
 - Popular approach on console
 - Partition all memory up front
- Load new level
 - Rewind level part only
- Reconfigure systems
 - Rewind both level, systems
- Fragmentation not possible



Linear Allocation

- Many games use linear allocators to achieve this kind of memory map
- Linear allocators basically sit on a pointer
 - Allocations just increment the pointer
 - To rewind, reset the pointer
- Very fast, but only suitable for POD data
 - No finalizers/destructors called
 - Used in Frostbite's renderer for command buffers

Linear Allocator Implementation

- Simplified C++ example
 - Real implementation needs checks, alignment
 - In retail build, allocation will be just a few cycles

```
1 class LinearAllocator {
2     // ...
3     u8 *allocate(size_t size) {
4         return m_ptr += size;
5     }
6     void rewind(u8 *ptr) {
7         m_ptr = ptr;
8     }
9     // ...
10    u8 *m_ptr;
11 };
```

Using Linear Allocation

- We're implementing FrogSystem
 - A new system tied to the level
 - Randomly place frogs across the level as the player is moving around
 - Clearly the Next Big Thing
- Design for linear allocation
 - Grab all memory up front



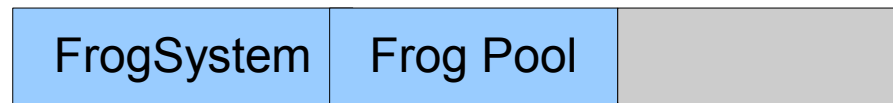
Mr FISK (c) FLT
Used with permission

FrogSystem - Linear Allocation

- Simplified C++ example

```
1 struct FrogInfo { ... };
2
3 struct FrogSystem {
4     // ...
5     int maxFrogs;
6     FrogInfo *frogPool;
7 };
8
9 FrogSystem* FrogSystem_init(LinearAllocator& alloc) {
10     FrogSystem *self = alloc.allocate(sizeof(FrogSystem));
11     self->maxFrogs = ...;
12     self->frogPool = alloc.allocate(sizeof(FrogInfo) * self->maxFrogs);
13     return self;
14 }
15
16 void FrogSystem_update(FrogSystem *system) {
17     // ...
18 }
```

Resulting Memory Layout



Allocation
Point

 POD Data

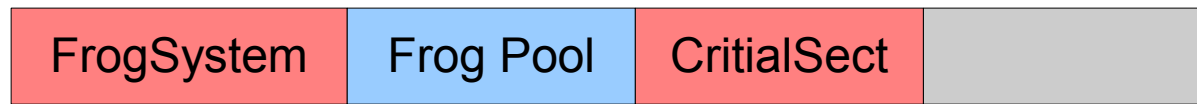
Linear allocation limitations

- Works well until we need resource cleanup
 - File handles, sockets, ...
 - Pool handles, other API resources
 - This is the “systems programming” aspect
- Assume frog system needs a critical section
 - Kernel object
 - Must be released when no longer used

FrogSystem – Adding a lock


```
1 class FrogSystem {
2     CriticalSection *m_lock;
3
4     FrogSystem(LinearAllocator& a)
5         // get memory
6         , m_lock((CriticalSection*) a.allocate(sizeof(CriticalSection)))
7         // ...
8     {
9         new (m_lock) CriticalSection; // construct object
10    }
11
12    ~FrogSystem() {
13        m_lock->~CriticalSection(); // destroy object
14    }
15 };
16
17 FrogSystem* FrogSystem_init(LinearAllocator& a) {
18     return new (a.allocate(sizeof(FrogSystem))) FrogSystem(a);
19 }
20
21 void FrogSystem_cleanup(FrogSystem *system) {
22     system->~FrogSystem();
23 }
```

Resulting Memory Layout



Allocation
Point

An arrow points from the text 'Allocation Point' to the boundary between the 'CritialSect' segment and the empty grey segment.

-  POD Data
-  Object with cleanup

Linear allocation limitations

- Code quickly drowns in low-level details
 - Lots of boilerplate
- We must add a cleanup function
 - Manually remember what resources to free
 - Error prone
 - In C++, we would rather rely on destructors

Scope Stacks

- Introducing Scope Stacks
 - Sits on top of linear allocator
 - Rewinds part of underlying allocator when destroyed
- Designed to make larger-scale system design with linear allocation possible
 - Maintain a list of finalizers to run when rewinding
 - Only worry about allocation, not cleanup

Scope Stacks, contd.

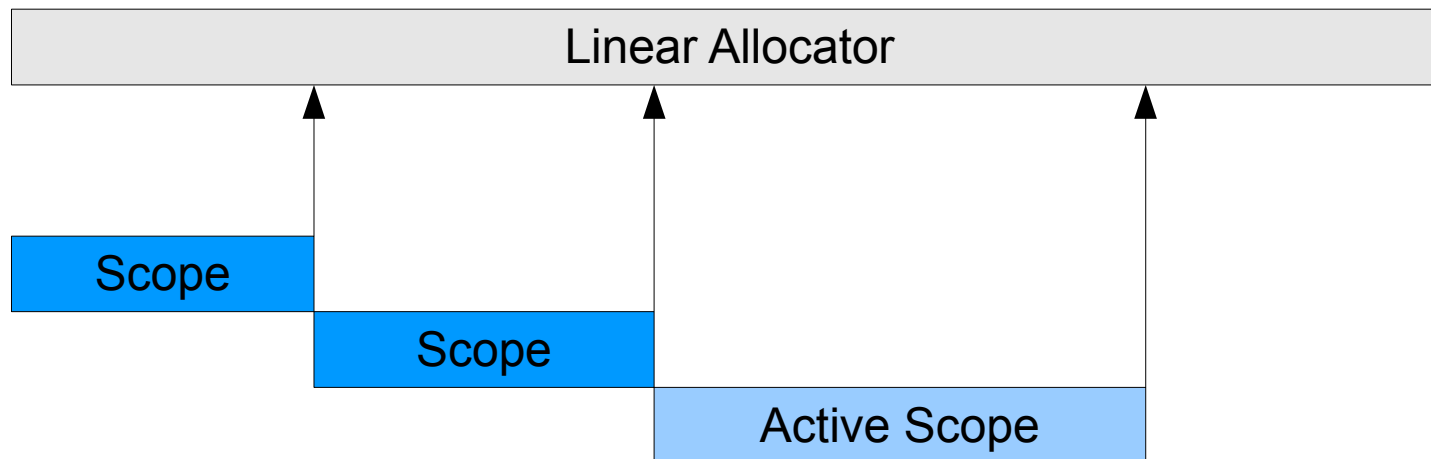
- Type itself is a lightweight construct

```
1 struct Finalizer {
2     void (*fn)(void *ptr);
3     Finalizer *chain;
4 };
5
6 class ScopeStack {
7     LinearAllocator& m_alloc;
8     void *m_rewindPoint;
9     Finalizer *m_finalizerChain;
10
11     explicit ScopeStack(LinearAllocator& a);
12     ~ScopeStack(); // unwind
13
14     template <typename T> T* newObject();
15     template <typename T> T* newPOD();
16 };
17
```

Scope Stacks, contd.

- Can create a stack of scopes on top of a single linear allocator
 - Only allocate from topmost scope
- Can rewind scopes as desired
 - For example init/systems/level
 - Finer-grained control over nested lifetimes
- Can also follow call stack
 - Very elegant per-thread scratch pad

Scope Stack Diagram



Scope Stack API

- Simple C++ interface
 - `scope.newObject<T>(...)` - allocate object with cleanup (stores finalizer)
 - `scope.newPod<T>(...)` - allocate object without cleanup
 - `scope.alloc(...)` - raw memory allocation
- Can also implement as C interface
 - Similar ideas in APR (Apache Portable Runtime)

Scope Stack Implementation

- `newObject<T>()`

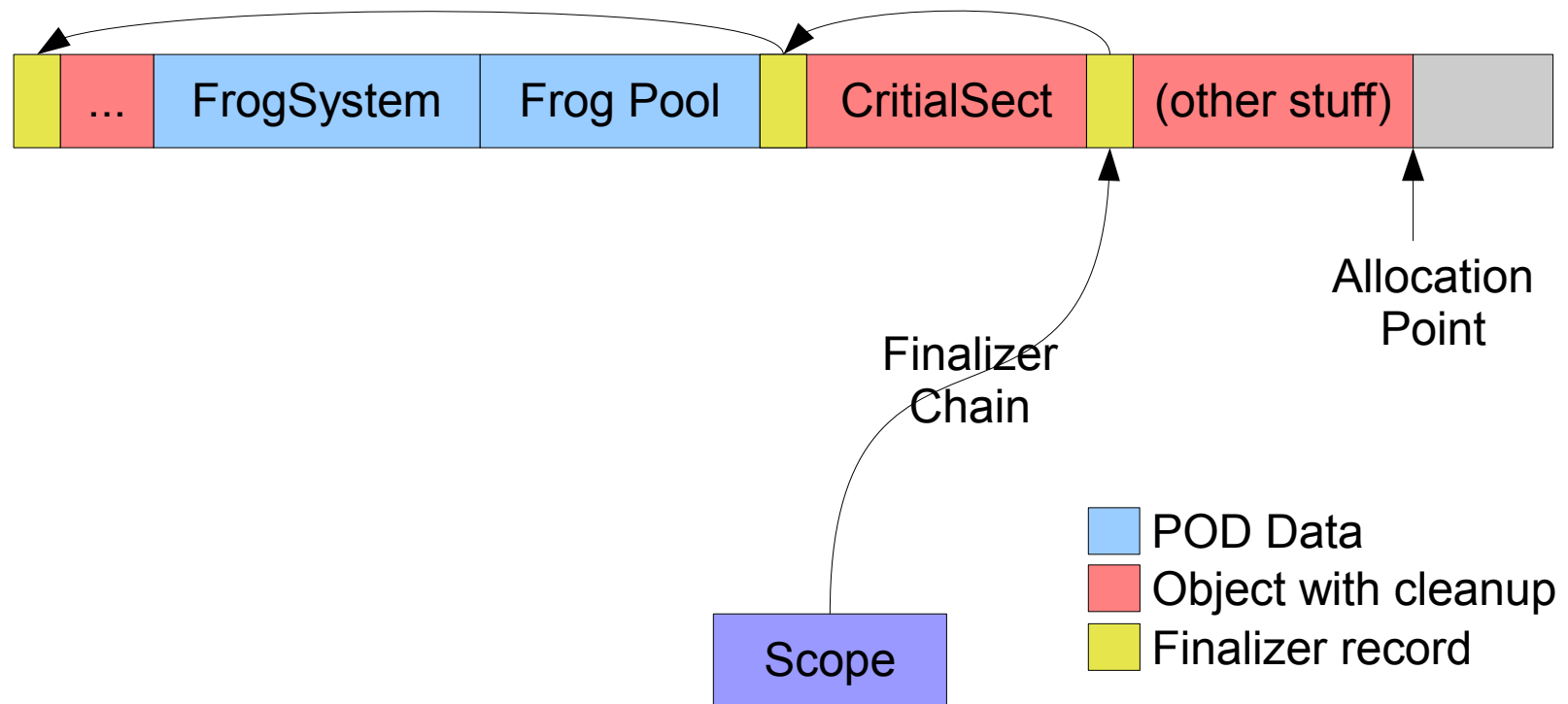
```
1  template <typename T>
2  void destructorCall(void *ptr) {
3      static_cast<T*>(ptr)->~T();
4  }
5
6  template <typename T>
7  T* ScopeStack::newObject() {
8      // Allocate memory for finalizer + object.
9      Finalizer* f = allocWithFinalizer(sizeof(T));
10
11     // Placement construct object in space after finalizer.
12     T* result = new (objectFromFinalizer(f)) T;
13
14     // Link this finalizer onto the chain.
15     f->fn = &destructorCall<T>;
16     f->chain = m_finalizerChain;
17     m_finalizerChain = f;
18     return result;
19 }
```

FrogSystem – Scope Stacks

- Critical Section example with Scope Stack

```
1 class FrogSystem {
2     // ...
3     CriticalSection *m_lock;
4
5     FrogSystem(ScopeStack& scope)
6         : m_lock(scope.newObject<CriticalSection>())
7         // ...
8     {}
9
10    // no destructor needed!
11 };
12
13 FrogSystem* FrogSystem_init(ScopeStack& scope) {
14     return scope.newPod<FrogSystem>();
15 }
```

Memory Layout (with context)



Scope Cleanup

- With finalizer chain in place we can unwind without manual code
 - Iterate linked list
 - Call finalizer for objects that require cleanup
 - POD data still zero overhead
 - Finalizer for C++ objects => destructor call

Per-thread allocation

- Scratch pad = Thread-local linear allocator
 - Construct nested scopes on this allocator
 - Utility functions can lay out arbitrary objects on scratch pad scope

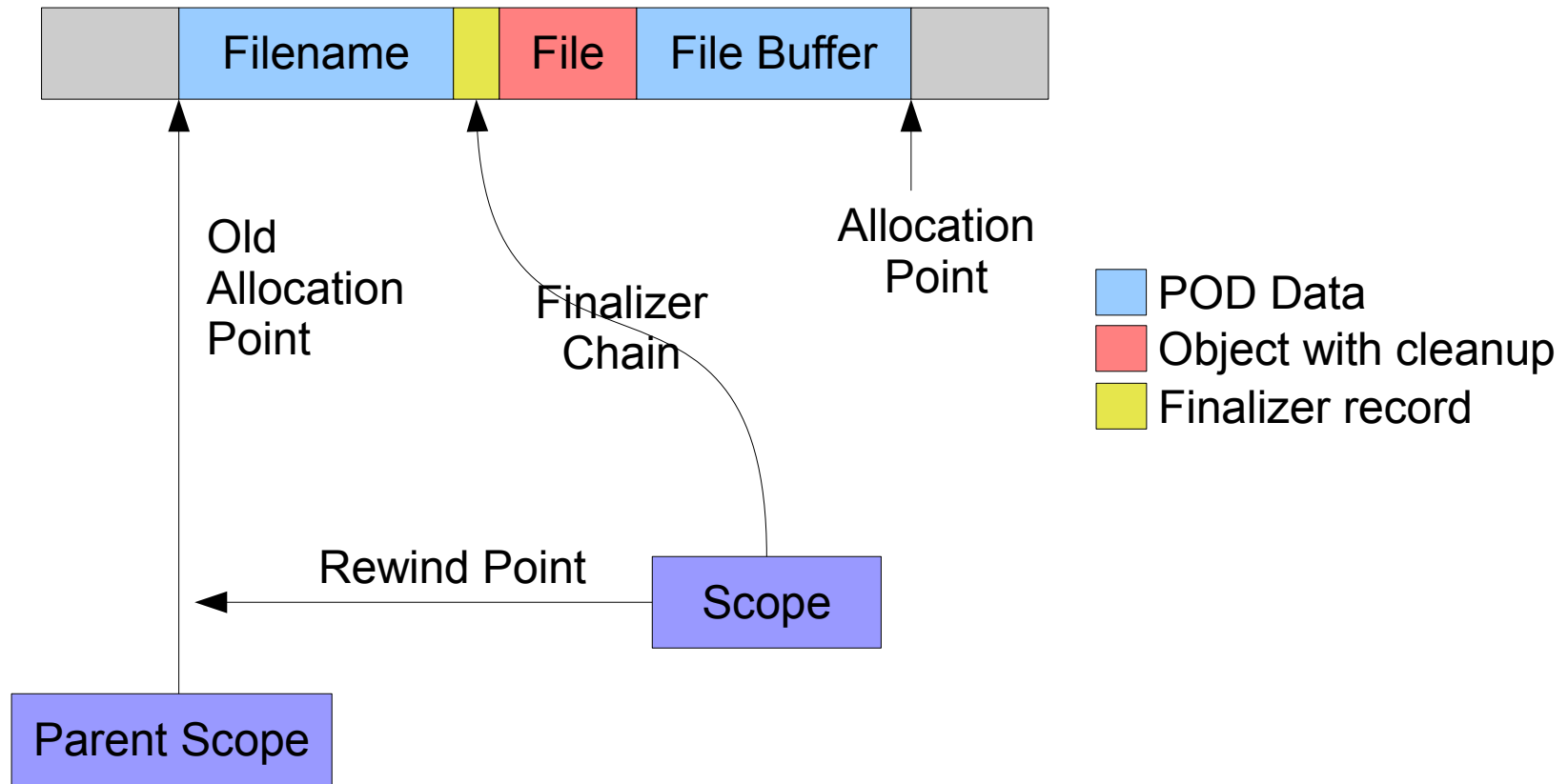
```
1 class File; // next slide
2
3 const char *formatString(ScopeStack& scope, const char *fmt, ...);
4
5 void myFunction(const char *fn) {
6     ScopeStack scratch(tls_allocator);
7     const char *filename = formatString(scratch, "foo/bar/%s", fn);
8     File *file = scratch.newObject<File>(scratch, filename);
9
10    file->read(...);
11
12    // No cleanup required!
13 }
```

Per-thread allocation, contd.

- File object allocates buffer from designed scope
 - Doesn't care about lifetime – its buffer and itself will live for exactly the same time
 - Can live on scratch pad without knowing it

```
1 class File {
2 private:
3     u8 *m_buffer;
4     int m_handle;
5 public:
6     File(ScopeStack& scope, const char *filename)
7         : m_buffer(scope.alloc(8192))
8         , m_handle(open(filename, O_READ))
9     {}
10
11     ~File() {
12         close(m_handle);
13     }
14 };
```

Memory Layout: Scratch Pad



PIMPL

- C++ addicts can enjoy free PIMPL idiom
 - Because allocations are essentially “free”; PIMPL idiom becomes more attractive
 - Can slim down headers and hide all data members without concern for performance

Limitations

- Must set upper bound all pool sizes
 - Can never grow an allocation
 - This design style is classical in games industry
 - But pool sizes can vary between levels!
 - Reconfigure after rewind
- By default API not thread safe
 - Makes sense as this is more like layout than allocation
 - Pools/other structures can still be made thread safe once memory is allocated

Limitations, contd.

- Doesn't map 100% to C++ object modeling
 - But finalizers enable RAII-like cleanup
- Many traditional C++ tricks become obsolete
 - Pointer swapping
 - Reference counting
- Must always think about lifetime and ownership when allocating
 - Lifetime determined on global level
 - Can't hold on to pointers – unwind = apocalypse
 - Manage on higher level instead

Conclusion

- Scope stacks are a system programming tool
 - Suitable for embedded systems with few variable parameters – games
- Requires planning and commitment
- Pays dividends in speed and simplicity
 - Same pointers every time – debugging easier
 - Out of memory analysis usually very simple
- Either the level runs, or doesn't run
 - Can never fail half through

Links

- Toy implementation of scope stacks
 - For playing with, not industrial strength
 - <http://pastebin.com/h7nU8JE2>
- “Start Pre-allocating And Stop Worrying” - Noel Llopis
 - <http://gamesfromwithin.com/start-pre-allocating-and-stop-worrying>
- Apache Portable Runtime
 - <http://apr.apache.org/>

Questions

Bonus: what about..

- ..building an array, final size unknown?
 - Standard approach in C++: STL vector push
 - Instead build linked list/dequeue on scratch pad
 - Allocate array in target scope stack once size is known
- ..dynamic lifetime of individual objects?
 - Allocate object pool from scope stack
 - Requires bounding the worst case – a good idea for games anyway