HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

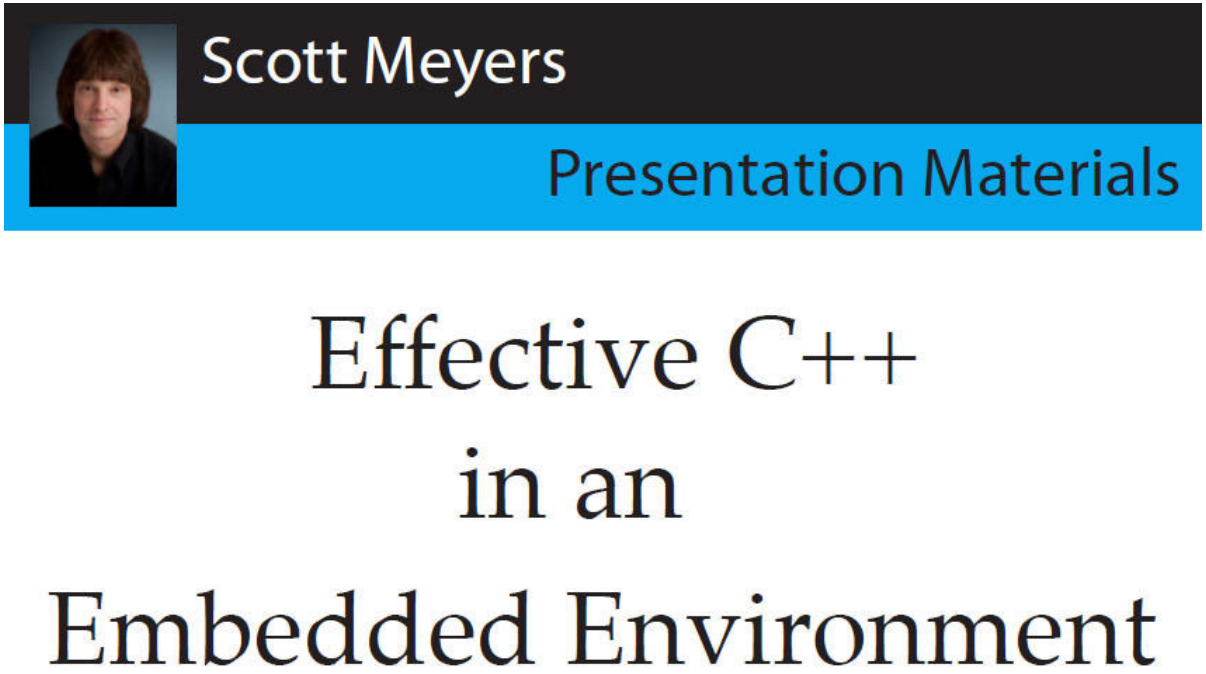
Embedded Software Engineering 2
Dynamic Memory Management (DMM)

Prof. Reto Bonderer
HSR Hochschule für Technik Rapperswil
reto.bonderer@hsr.ch

April 2020

Effective C++ in an Embedded Environment

Die meisten der folgenden Informationen stammen aus einem Vortrag von Scott Meyers



Dynamische Speicherverwaltung

Scott Meyers:

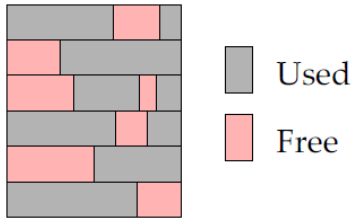
- Embedded developers often claim heap management isn't an issue:
  - Client: "We don't have a heap."
  - Me: "You're right. You have five heaps."
- Dynamic memory management is present in many embedded systems.
  - Even if malloc/free/new/delete never called.
- Key indicator:
  - Variable-sized objects going in fixed-size pieces of memory.
  - E.g., event/error logs, rolling histories, email messages, etc.

Dynamic Memory Management

Four common worries

- **Speed**
  - Are new/delete/malloc/free fast enough?
  - How much variance, i.e. how deterministic?
- **Fragmentation**
  - Will heap devolve into unusably small chunks?
    - This is external fragmentation
- **Memory leaks**
  - Will some allocations go undeallocated?
- **Memory exhaustion**
  - What if an allocation request can't be satisfied?

Each concern can be addressed.



© Scott Meyers

Grundsätzliches zu Dynamic Memory Management

- Es gibt unterschiedliche Strategien, wie DMM umgesetzt werden kann
  - new/delete/malloc/free ist nicht die einzige Variante
- Die "übliche" Umsetzung mit `new/delete` beinhaltet die Gefahr der Fragmentierung und dadurch eines nicht deterministischen Verhaltens
- Fragmentierung entsteht jedoch nur durch fortlaufendes `new/delete`
- Wenn `new` nur beim Aufstarten (welches meist nicht zeitkritisch ist) durchgeführt wird, `delete` erst beim Herunterfahren, dann besteht kein Fragmentierungsproblem
- Unterschiedliche Konfigurationen können damit sehr elegant gelöst werden

A Survey of Allocation Strategies

Each less general than malloc/free/new/delete.

- Typically more suited to embedded use.

We'll examine:

- Fully static allocation
- LIFO allocation
- Pool allocation
- Block allocation
- Region allocation
  - An optimization that may be combined with other strategies.

Fully Static Allocation

No heap. Objects are either:

- On the stack: Local to a function.
- Of static storage duration:
  - At global scope.
  - At namespace scope.
  - static at file, function, or class scope.

Useful when:

- Exact or maximum number of objects in system statically determinable.

Fully Static Allocation (cont'd)

“Allocation” occurs at build time. Hence:

- Speed: essentially infinite; deterministic.
- External Fragmentation: impossible.
- Memory leaks: impossible.
- Memory exhaustion: impossible.

But:

- Initialization order of static objects in different translation units (TUs) indeterminate.

“Heap Allocation”

Two common meanings:

- Dynamic allocation outside the runtime stack.
- *Irregular* dynamic allocation outside the runtime stack.
  - Unpredictable numbers of objects.
  - Unpredictable object sizes.
  - Unpredictable object lifetimes.

We’ll use the first meaning.

- The second one is just the most general (i.e., hardest) case of the first.

User-controlled non-heap memory for multiple variable-sized objects entails heap management:

```
uint8_t buffer[someSize];  // this is basically a heap; create/destroy multiple
...                        // different-sized objects in buffer
```

The C++ Memory Management Framework

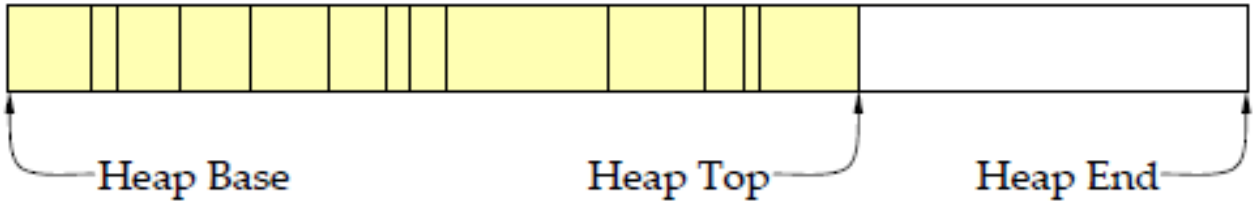
User-defined memory management typically built upon:

- User-defined versions of malloc/free
- User-defined versions of operator new/new[], operator delete/delete[]
- new handlers:
  - Functions called when operator new/new[] can’t satisfy a request.

Here we focus on allocation strategies suitable for embedded systems.

Example: LIFO Heap Allocation

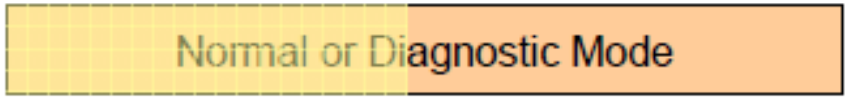
Dynamic allocation is strictly LIFO (like a stack).



Easy way to implement a "union" for multiple-mode operations:

- E.g., a system in "normal" or "diagnostic" mode.
  - Static allocation requires the *sum* of the two modes' memory needs.

  - LIFO allocation only the *maximum* of the modes' needs.



© Scott Meyers

LIFO Heap Allocation, First Cut

```
class LIFOAllocator {                    // provides behavior of new/delete via allocate/deallocate
public:
  LIFOAllocator(uint8_t* heapAddr, std::size_t heapSize)
  : heapBase(heapAddr), heapEnd(heapAddr+heapSize), heapTop(heapAddr)
  {}
 void* allocate(std::size_t sz) throw (std::bad_alloc); // not shown
 void deallocate(void* ptr, std::size_t sz) throw ();   // ditto
private:
  uint8_t* const heapBase;
  uint8_t* const heapEnd;
  uint8_t* heapTop;
};
```

- allocate/deallocate behave like class-specific new/delete.
- Pointer data member ⇒ copying functions should be declared.
- If LIFOAllocator templatized, ctor params could be template params.
  - The MMIO section has an example.

LIFO Heap Allocation, First Cut (cont'd)

Classes can easily build custom new/delete using LIFOAllocator:

```
uint8_t heapSpace[heapSpaceSize];           // memory for heap

LIFOAllocator customAllocator(heapSpace,    // typically at global scope
                              heapSpaceSize);

void* Widget::operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return customAllocator.allocate(bytes);
}

void Widget::operator delete(void* ptr, std::size_t size) throw ()
{
  customAllocator.deallocate(ptr, size);
}
```

LIFO Heap Allocation

- Speed: extremely fast; deterministic.
  - Assuming you don't run out of memory
- External Fragmentation: possible, but easy to detect.
- Memory leaks: possible, easy to detect.
- Memory exhaustion: possible.

Pool Allocation

Heap allocations are all the same size.

- Typically because all heap objects are one size.
  - Well-suited for class-specific allocators.
- Can also work when all heap objects are nearly the same size.
  - Then all allocations are the size of the largest objects.

Basic approach:

- Treat heap memory as an array.
  - Each element is the size of an allocation unit, therefore no need to store the size of each allocation.
- Unallocated elements are kept on a *free list*.
- Allocation/deallocation is a simple list operation:
  - Removing/adding to the front of the free list.

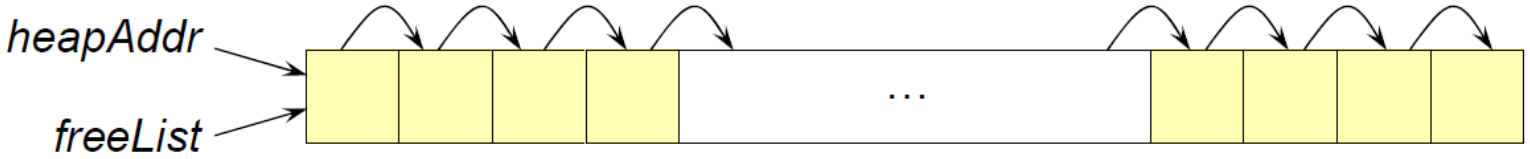
Pool Allocation (cont'd)

```
template<std::size_t elementSize>
class PoolAllocator {
public:
  PoolAllocator(uint8_t* heapAddr, std::size_t heapSize);   // on next page
  void* allocate(std::size_t sz) throw (std::bad_alloc);    // coming soon
  void deallocate(void* ptr, std::size_t sz) throw ();      // ditto
private:
  union Node {                                  // pool element
              uint8_t data[elementSize];        // when in use
              Node* next;                       // on free list
  };
  Node* freeList;
};
```

- Pointer data member ⇒ copying functions should be declared.
- If PoolAllocator untemplatized, template param could be ctor param.
- Ideally, we'd ensure that `elementSize > 0`, better: `>= sizeof(Node*)`.

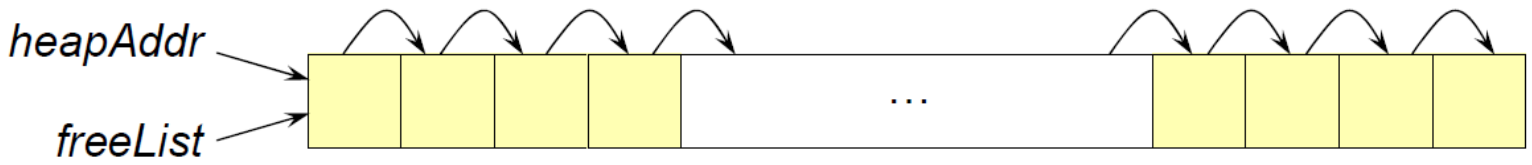
PoolAllocator Constructor

```
template<std::size_t elementSize>
PoolAllocator<elementSize>::PoolAllocator(uint8_t* heapAddr,
                                          std::size_t heapSize)
  : freeList(reinterpret_cast<Node*>(heapAddr))
{
  const std::size_t nElems = heapSize / sizeof(Node);
  for (std::size_t i = 0; i < nElems-1; ++i)   // link array elements
    freeList[i].next = &freeList[i+1];
  freeList[nElems-1].next = nullptr;           // nullptr from and after C++11
}
```



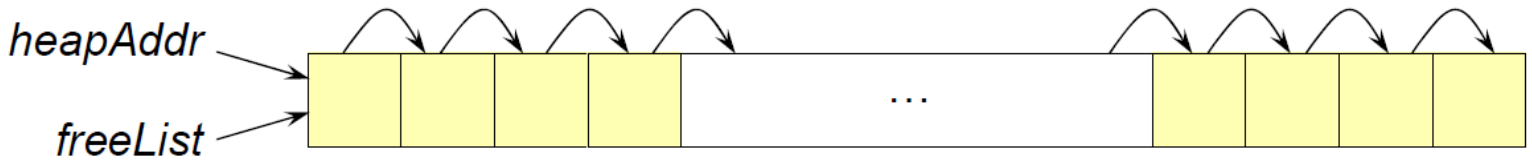
PoolAllocator::allocate()

```
template<std::size_t elementSize>
void* PoolAllocator<elementSize>::allocate(std::size_t bytes) throw (std::bad_alloc)
{
  if (bytes != elementSize)
    return ::operator new(bytes);
  if (freeList != nullptr)
  {
    void* pMem = freeList;              // alignment?
    freeList = freeList->next;
    return pMem;
  }
  else
    throw std::bad_alloc();
}
```



PoolAllocator::deallocate()

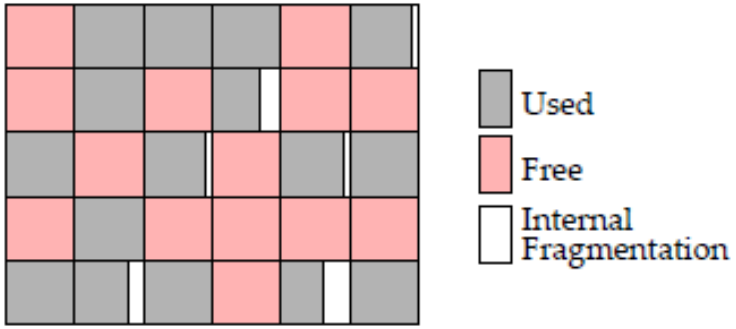
```
template<std::size_t elementSize>
void PoolAllocator<elementSize>::deallocate(void* ptr,
                                            std::size_t size) throw ()
{
  if (ptr == nullptr)
    return;
  if (size != elementSize)
  {
    ::operator delete(ptr);
    return;
  }
  Node* p = static_cast<Node*>(ptr);
  p->next = freeList;
  freeList = p;
}
```



PoolAllocator::allocate()

Variation: allow bytes <= elementSize, i.e., that the request fits.

- More flexible, but can lead to internal fragmentation.

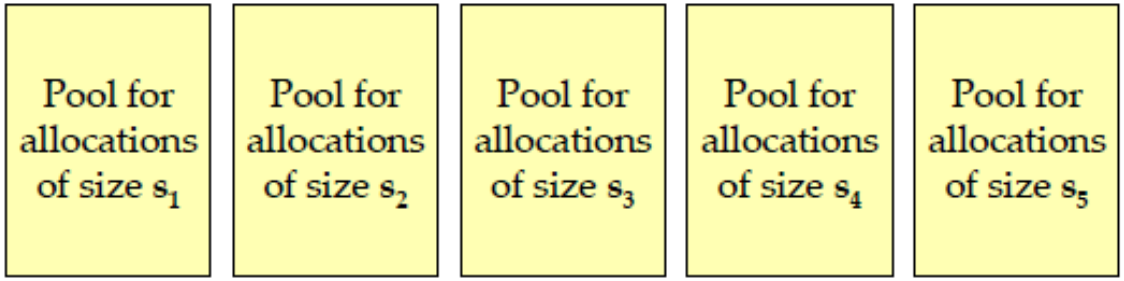


Pool Allocation

- Speed: extremely fast; deterministic.
  - Assuming
    - No wrong-sized requests
    - You don't run out of memory
- External Fragmentation: impossible
- Memory leaks: possible
- Memory exhaustion: possible.

Block Allocation

Essentially a set of pools with different element (block) sizes:



n-byte requests handled by first pool with size ≥ n and non-null free list.

Useful when:

- Allocations needed for a relatively small number of object sizes.
  - Otherwise internal fragmentation ⇒ wasted memory.

Many RTOSes offer native support for block allocation.

Block Allocation

- Speed: fast; nearly deterministic (and boundable).
  - Assuming
    - No requests larger than handled by the largest-chunk pool.
    - You don't run out of memory
  - Speed isn’t totally deterministic, because you may need to examine multiple pools to find one with sufficient free memory.
- External Fragmentation: impossible
- Memory leaks: possible
- Memory exhaustion: possible.

General Variable-Sized Allocation

What new/delete/malloc/free already do.

- Desirable only if vendor-supplied routines unacceptable.

Possible motivations:

- Detect overruns/underruns.
- Gather heap usage data.
  - Size and lifetime distributions, temporal usage patterns, etc.
- Support data structure clustering.
- Avoid thread-safety penalty.
  - ST applications.
  - Thread-local allocators in MT applications.

Region Allocation

An optimization for when memory for all of a heap's objects can be released at once.

- Clients call a region member function at the appropriate time.
  - Faster than deallocating each object's memory individually.
- Common with LIFO allocators, but compatible with pools, blocks, etc.
- operator delete for individual objects a no-op, hence very fast.
  - Can still use delete operator to invoke destructors:

```
delete p;       // invoke *p's dtor, then operator delete on p;
                // if *p in a region, operator delete is a no-op
```

Summary: Dynamic Memory Management

- Many embedded systems include dynamic memory management.
- Key issues are speed, fragmentation, leaks, and memory exhaustion.
- LIFO is fast and w/o fragmentation, but object lifetimes must be LIFO.
- Pools are fast and w/o fragmentation, but object sizes are limited.
- Block allocation is essentially multiple pool allocators.
- Regions excel when all heap objects can be released simultaneously.