Is operator new redundant in C++11?

Niall Cooling, Feabhas Ltd.

Dynamic Memory

Why useful?

Depending on how we define program variables affects how we allocate the RAM. This placement of the program variables affects the memory's scope & lifetime; scope being where in the program we can directly access it for read/write operations and lifetime being for how long that memory is allocated for.

When allocation RAM in a C/C++ program we have three options:

1. Static
2. Stack
3. Free Store[1]/Heap

Static memory is where objects are defined at file scope (we often refer to these as "global" variables) and any class based members declared as static.[2] The scope of the memory is either external or internal depending on the use of unnamed namespaces. The lifetime of this memory is for the entirety of the program execution; in a typical embedded system these will have absolute addresses in RAM. We know the memory requirements at link time.

The memory for objects local to a function is typically allocated from the stack. The scope of the memory is from definition to the end of the enclosing block. The lifetime of the memory (how long it is valid to access it for) is the duration of the enclosing block. When we exit a block the memory is normally reclaimed by popping the stack. Historically, in C, these were known as autos (for automatics; as in the memory is automatically allocated and deallocated) but with the advent of the C++11 standard we should not use the term "auto" any more[3].

Stack memory is dynamic memory allocation; the benefit being that we only allocate memory when we need it, reducing the overall RAM requirements compared to if all memory was statically allocated. However, as the memory is managed by the C++ runtime system we can say this is "constrained" dynamic memory allocation.

So why do we need a further model for memory allocation? There are cases where ideally we need to lifetime of the memory to persist beyond a function, but trying to pre-allocate enough memory statically may end up potentially wasting a lot RAM if that is not required.

Take, as a simple example, managing a set of tracks for a GPS unit. The number of data points required per track isn't known in advance as it is dependent on the combination of track length and time duration of measurement. If we look to statically allocate the RAM for a list we run the

---

[1] new allocates from the Free Store whereas malloc allocates from the heap. However from hereon the two terms will be used synonymously.
[2] I'm not forgetting function based objects defined as static, but that's poor C++
[3] more on this later

risk of (a) not having enough memory for a track, or (b) wasting memory for short tracks. Either way fixed allocation is difficult and an easier model is to use dynamic data structures (DDS). In C these are typically hand-crafted, but in C++ many of the common DDSs are supported by the STL (lists, sets, maps, etc.). These DDSs, by default, use dynamic memory allocation in the background.

Unconstrained dynamic memory allocation[4] is, of course, supported in C++ through the new and delete functions (replacing the C malloc/free API).

MISRA-C++

However, with all the potential benefits and simplifications of using dynamic memory allocation, many embedded coding standards ban the use of unconstrained dynamic memory allocation. This is typified by the MISRA-C++ coding guidelines:

> ***Rule 18-4-1 (Required) Dynamic heap memory allocation shall not be used.***

The guideline has a rationale of why of it bans the use of heap memory usage.

Dangers associated with Dynamic Memory Usage

The dangers associated with the use of dynamic memory allocation can be broken down into three major groups.

Exhaustion

This is probably the most obvious: what happens if we run out of heap memory? How does the program deal with this run-time failure? In high integrity systems the potential for memory exhaution is normally considered unacceptable behavior.

Fragmentation

Fragmentation of the heap occurs because memory requests must be allocated as a single contiguous block. As blocks are allocated and released the heap is broken up in to different free memory blocks. A memory manager will concatenate two adjoining free blocks in to a single larger block, but if the free memory isn't adjoining they have to be treated as two singe fixed size blocks (creating 'holes' in the heap). When a new block is requested the memory manager will use a given rule to allocate from the available free memory (best-fit, first-fit, worst-fit, etc.); this can further accelerate fragmentation.

In itself fragmentation isn't a major problem, but can lead to two further problems:

- it can compromise the time to dynamically allocate or free memory, and;
- it can lead to premature memory exhaustion as no single free block is large enough to accommodate the requested memory, even though the overall free memory may exceed the request

Incorrect use of the API

There are a number of cases where the misuse of the delete function can cause major runtime problems:

- new[] and delete;
  a well known mistake is to call delete on memory allocated using new[]. This leads to a silent memory leak.

[4] From here on assume any reference to dynamic memory means unconstrained

- Delete non-dynamic memory;
  if a pointer is passed to the delete function that does not contain the address of previously dynamically allocated memory then the resulting behavior is undefined, but most likely the program will crash.
- Memory Leaks;
  Probably the most common bug in programs that allow unconstrained dynamic memory allocation. This is the simple case of memory that has been allocated through a call to new never being deleted.

Addressing the issues

Exhaustion

We cannot escape the fact that RAM sizes in embedded systems are forever getting bigger. Even many "small" systems may have many tens of kilobytes of RAM, and it is not uncommon to see larger systems with megabytes of RAM (especially those running Linux). The probability of exhausting memory, therefore, is diminished in many embedded systems.

Should we happen to exhaust memory C++ uses the exception model to report and manage such an occasion by throwing the std::bad_alloc exception. The benefit of the exception model is that it gives us a more formal and controlled technique for dealing with, and hopefully, recovering from such events.

Fragmentation

Many small C/C++ runtime systems use a very simple allocation scheme, for example where the heap is managed as a single-linked list of free blocks held in increasing address order. The allocation policy is first-fit by address. This sort of implementation has very low overheads, but the performance cost of allocation/deallocation grows linearly with the number of free blocks. Typically the smallest block that can be allocated is four bytes and there is an additional overhead per allocation (of four bytes).

The onset and, thus, problem of fragmentation is affected by (a) the allocation scheme, and (b) the variation and timing of the allocation/deallocation of different size objects. A lot of research has been applied over the years specifically to address fragmentation of the heap.

The most common approach in embedded systems, especially among Real-Time Operation Systems, is using a pool-based fixed-block allocation scheme. Here the heap is pre-partitioned into a set of pools. Each pool is made up of a collection of equal size fixed blocks, e.g. a 1kB pool may be partitioned into 64 x 16-byte blocks, whereas another 1kB pool may be partitioned into 8 x 128-byte blocks. A memory allocation request is fulfilled by allocating a block from the best-fit pool (e.g. a 12-byte request would allocate a block from the 16-byte pool). When that block is deallocated all 16-bytes are freed, thus eliminating fragmentation within the pool. Naturally this scheme can be inefficient in terms of heap utilization (e.g. a 20-byte block would be allocated within a 128-byte block), so the choice of pool/block-size is critical.

In larger systems there exist a number of different memory management schemes including the Slab allocator, the SLOB allocator and the buddy memory allocator, all focusing on mechanism to address fragmentation.

However, the C++ programmer has additional tools at their disposal, most notably the ability to overload the new and delete operators on a class-by-class basis. The difficulty of fixed-block allocation schemes is matching the requested size to the block size; ideally these would be an

exact match. By overloading new/delete it is "relatively" straightforward to implement a fixed-block pool system where a pool is dedicated to a class that may require dynamic memory allocation and the block-size is an exact match to the class size.

As mentioned earlier, the STL Dynamic Data Structures, such as the list, use the heap for memory allocation by default. However, each instance of a container can be passed an allocator class that replaces the default heap manager. As with the class-new model, with a small amount of work an efficient non-fragmenting allocator can be implemented or you can reuse one of the many open implementations on the Internet, e.g. Loki's SmallObject Allocator.

Incorrect use of the API

Exhaustion and fragmentation are runtime consequences of using dynamic memory allocation. They can both occur in a "correct" program. However the final set of problems all relate to the misuse of the delete function in connection to new'ed memory.

In modern C++ it is preferable to use the STL rather than calling array-new (new[]). The STL vector is a far better mechanism than the raw array, and with appropriate use of the 'reserve' function memory can be managed efficiently.

With the advent of C++11, the obsoleting of the raw array has moved on significantly with the addition to the STL of std::array and the ability to initialize containers at their definition.

The last two points; freeing non-dynamic memory and memory leaks can both be addressed by eliminating the direct call to delete from application code.

Smart pointers

Concept

Smart pointers (also referred to as Managed Pointers) use a very simple technique to ensure that memory leaks can not happen. For example a simple smart pointer implementation:

```
class EntityPtr
{
public:
    EntityPtr(Entity* p);
    ~EntityPtr();
private:
    Entity* ptr;
};

EntityPtr::EntityPtr(Entity* p):ptr(p)
{
}

EntityPtr::~EntityPtr()
{
    delete ptr;
}
```

The use of the smart pointer is thus:

```
void f(int val)
{
  EntityPtr  ptr(new Entity(val));
  // work with memory
  // return (planned or unplanned)
}
```

The advantage of this model is that the memory is released when the smart pointer's lifetime ends (here on exit from the function); there is no possibility of a memory leak.

However for a smart pointer to replace a raw pointer it must also overload an number of other member functions, specifically:

- operator*() return a reference of the object
- operator->() returns object's address, -> applied
- operator=() copy-suppress, pass-ownership, reference-counting
- copy ctor as per operator=

The following code is a template smart pointer that supresses copying.

```
template <typename T>
class SmartPtr
{
public:
    SmartPtr(T* p);
    ~SmartPtr();
    T* operator->() const { return ptr; }
    T& operator*()  const { return *ptr;}
    operator T*()   const { return ptr; }
private:
    SmartPtr& operator=(const SmartPtr&);
    SmartPtr(const SmartPtr&);
    T* ptr;
};

freeFunc(Entity* p);
freeFunc(Entity  e);

void f(int val)
{
  SmartPtr<Entity> ptr(new Entity(val));

  ptr->func();
  freeFunc(ptr);
  freeFunc(*ptr);
}
```

std::auto_ptr

A smart pointer existed as part of the original C++ Standard Library (C++98), called std::auto_ptr. auto_ptr objects have the peculiarity of *taking ownership* of the pointers assigned to them: An auto_ptr object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed. The destructor does this by calling operator delete automatically.

Therefore, no two auto_ptr objects should *own* the same element, since both would try to destruct them at some point. When an assignment operation takes place between two auto_ptr objects, *ownership* is transferred, which means that the object losing ownership is reset to no longer point to the element (it is set to the *null pointer*).

However this 'transfer of ownership' lead to problems when trying to use auto_ptr with STL containers as it does not meet the basic requirements for container types (i.e. the sort algorithm makes internal copies of objects leading to transfer of ownership to the copies).

Boost

Due to the problematic nature of auto_ptr, the Boost library[5] developed in parallel a set of alternative smart pointers, amoung them the shared_ptr.

The shared_ptr class template, like the auto_ptr, stores a pointer to a dynamically allocated object. The object pointed to is guaranteed to be deleted when the last shared_ptr pointing to it is destroyed or reset. However, shared_ptr uses a reference counter implementation, so that shared_ptr meets the CopyConstructible and Assignable requirements of the C++ Standard Library, and so can be used in standard library containers. Comparison operators are supplied so that shared_ptr works with the standard library's associative containers.

The shared_ptr can lead to problem of cyclic dependency, but this is addressed by another smart pointer; the weak_ptr. I won't go into the details of the weak_ptr here.

In 2005 a C++ Technical Report was published (TR1) which proposed additions to the C++ standard library for the C++98[6] language standard. Part of this included the Boost smart pointers smart_ptr and weak_ptr.

C++11

With the publication of ISO/IEC 14882:2011 a new iteration of the C++ standard was born. As part of the new standard most of TR1 was incorporated.; specifically shared_ptr and weak_ptr both became part of the Standard Library.

std::shared_ptr

A simple example using the std::shared_ptr[7]:

```
#include <memory>
```

[5] Boost provides free peer-reviewed portable C++ source libraries

[6] Correctly it I should say C++03, as there was a 2003 update to the original C++ 1998 standard, but from a programmers perspective they are one in the same.

[7] you may need <tr1/memory> and std::tr1::shared_ptr depending on your compilers support for C++11

```
void f()
{
   std::shared_ptr<Alarm> pA(new Alarm(1,true));
   // automatically deleted on exit
}
```

As part of the development of the shared_ptr in Boost, there were repeated requests from the user community for a factory function that creates an object of a given type and returns a shared_ptr to it. Besides wanting it for convenience and style, such a function offers an exception safe and considerably faster implementation because it can use a single allocation for both the object and its corresponding control block, eliminating a significant portion of shared_ptr's construction overhead. This eliminates one of the major efficiency complaints about shared_ptr.

```
#include <memory>

void f()
{
   // std::shared_ptr<Alarm> pA(new Alarm(1,true));
   std::shared_ptr<Alarm> pA1 = std::make_shared<Alarm>(1,true);
   // automatically deleted on exit
}
```

So, with the advent of shared_ptr and make_shared as part of the C++11 standard we can see that the use of new can pretty much be eliminated from application code.

std::unique_ptr

"Not so fast!" I hear you cry; what if I don't want a shared pointer, what if I want a non-copyable pointer (unique ownership) ?

Included as part of the C++11 standard was the addition of another smart pointer std::unique_ptr:

- std::unique_ptr retains sole ownership of an object through a pointer, and
- std::unique_ptr destroys the pointed-to object when the unique_ptr goes out of scope.

unique_ptr is not copyable or copy-assignable, two instances of unique_ptr cannot manage the same object. A non-const unique_ptr can transfer the ownership of the managed object to another unique_ptr using the new C++11 move constructor.

```
#include <memory>

struct X
{
   void op();
};

int main()
{
  std::unique_ptr<X> p1(new X);    // NB: You can't do:
                                   // std::unique_ptr<X> p1 = new X;
  p1->op();
  std::unique_ptr<X> p2(p1);            // ERROR - Copy construction
  std::unique_ptr<X> p3(std::move(p1)); // OK - move constructor called

}
```

Notice we're back to using raw new again! Unfortunately C++11 did not include a make_unique function – doh. Most industry experts agree that this is an oversight and will be address through a Technical Report (TR2 ?).

There are already a number of example implementations for make_unique on the web, using further C++11 additions, e.g.:

```
#include <iostream>
#include <memory>
#include <utility>

struct A
{
     A(int&& n) { std::cout << "rvalue overload, n=" << n << "\n"; }
     A(int& n)  { std::cout << "lvalue overload, n=" << n << "\n"; }
};

template<class T, class U> std::unique_ptr<T> make_unique(U&& u)
{
     return std::unique_ptr<T>(new T(std::forward<U>(u)));
}

int main()
{
     std::unique_ptr<A> p1 = make_unique<A>(2); // rvalue
     int i = 1;
     std::unique_ptr<A> p2 = make_unique<A>(i); // lvalue
}
```

Summary

Is new redundant?

Not quite; had a std::make_unique been included in C++11 you could, to all intense and purposes, eliminate the use of new in application code. But certainly the broader use of smart pointers eliminates the major problem of memory leaks associated with dynamic memory allocation.

However, we cannot ignore the serious issues of fragmentation and memory exhaustion. With appropriate use of fixed-block allocation we can eliminate most cases of fragmentation, nevertheless there always will be cases where we cannot predetermine the memory size required (e.g. a packet over a network).

Exhaustion is still our major impediment to using dynamic memory in real-time embedded systems. A good failure policy based around std::bad_alloc can address many of the issues, but in high integrity systems dynamic memory usage will remain unacceptable.