Memory & storage

Mixing malloc and new

When people talk about avoiding mixing new/delete with malloc()/free() they are right in doing so. They refer to creating an object with new for instance, then destroy the memory using free(); This kind of scenario has the consequence of not calling the destructor for the object.

When freeing an int or other primitive type, you don't have much to worry about, but if the object in question is a complex type there may be other issues to consider. If the object allocates memory itself, then that memory will not be freed if the destructor is in charge of deleting it.

A look at what the new and delete operator do internally reveals this:

- A standard new and delete combo.

```
MyClass *m = new MyClass();
//... Use m
delete m;
```

- What is close to really happening.

```
MyClass *m = ( MyClass* ) malloc( sizeof( MyClass ) );
m->MyClass(); //Call constructor.
//... Use m
m->~MyClass(); //Call destructor.
free( m );
```

You may have noticed that the above snippet calls both the constructor and destructor. Calling the destructor is fine and is commonly used in conjunction with the placement new and delete operators. Unlike the destructor call, the constructor call is invalid. A closer look at the highlighted portion of code reveals that it is actually an anonymous temporary being created, not a function call, which is not valid at that point in the code. The actual constructor call is initiated by the use of the operator new which includes the contruction definition (standard scenario snippet).

It also would not make much sense or be at all practical for the new operator to call the constructor. As an object can have more than one constructor, or a constructor with one or many parameters, new would have to know about not only the object type, but the constructor overloads, and all the parameters passed to them.

So there begins the challenge, how do you construct an object in memory that has already been allocated, consequently ruling out the use of new? The answer is placement new, which has the ability to construct an object at a pre-determined memory location; this implies the opreators do not allocate memory, but are supplied an allocated but uninitialised memoy chunk.

Here is a sketch which replicates the invalid snippet above, but uses the placement new operator to construct the object from memory allocated by malloc.

```
void *operator new( size_t size, void *ptr ){
  return ptr;
}

void operator delete( void *obj, void *alloc ){
  return;
}

class MyClass{
  public:
    MyClass( void ){}
    ~MyClass( void ){}
};

void setup(){

  void *chunk = malloc( sizeof( MyClass ) );

  //Placement new
  MyClass *m = new( chunk ) MyClass();

  //... Use m

  m->~MyClass(); //Call destructor.

  //Placement delete, this does nothing and can be omitted.
  ::operator delete( m, chunk );

  free( chunk );
}

void loop(){}
```

As placement new can be far more complex than standard new, it deserves its own article, so for further reading you can visit it here: Coming Soon!

There is no reason why you can't mix the two allocation schemes when used independently of each other, its not uncommon to want constructed objects and raw working space. Also if portability is important, there is no guarantee that malloc & new use the same heap/pool to store their allocations, which is another good reason to avoid mixing the scenarios. On the contrary however, placement new will happily construct an object anywhere its pointed to making the code above suitably portable.

- **Can I use new & delete with Arduino.**
- **How to use dynamic memory.**
- **Placement new & delete.**

If you found this article helpful, please rate this FAQ.
Unique solution ID: #1023
Author: Christopher Andrews
Last update: 2014-03-04 14:33