# Memory Management

Memory for any application is a crucial resource. No matter what type of application you are creating you will benefit from having good memory management. With some basic memory management in place you can know your memory distribution between systems, find memory leaks much easier, simplify memory alignment requirements, etc.

### Getting the Chaos Under Control

The first thing to do is to override the global `new` and `delete` functions. Doing this gives you a starting point for all your memory management. All calls to new and delete will be handled by your custom functions.

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

void delete (void* mem)
{
    free(mem);
}

// Don't forget the array version of new/delete
void* operator new[](size_t size)
{
    return malloc(size);
}

void delete (void* mem)
{
    free(mem);
}
```

## Custom Versions of `new` and `delete`

It would be very nice to find out who makes an allocation request. From the code doing the allocation it would sometimes be very nice to be able to also specify extra information such as the alignment needed for the memory block. Aligned memory in particular is a pain to work with unless it is supported by the memory allocator. If you have a class that needs to be 16-bytes long it will be messy. Instead you can create a class to keep track of the memory usage and override the `new` and `delete` operators:

```
// We use a class to represent the alignment to avoid any code
// situations where 'new (0x12345678) MyClass()' and
// 'new ((void*)0x12345678) Myclass()' might cause a placement
// construction instead of construction on an aligned memory
// address.
class Align
{
public:
    explicit Align(int value) : m_value(value)
    {
    }
    int GetValue() const
    {
       return m_value;
    }
private:
    int m_value;
```

```
};

// Overridden 'normal' new/delete
void* operator new (size_t size);
void* operator new[] (size_t size);
void operator delete( void* mem );
void operator delete[]( void* mem );

// Aligned versions of new/delete
void* operator new[] (size_t size, Align alignment);
void* operator new (size_t size, Align alignment);
void operator delete (void* mem, Align alignment);
void operator delete[] (void* mem, Align alignment);
```

Which is called like:

```
// Just allocate with an extra argument to 'new'
MyClass* pMyClassInst = new (Align(16)) MyClass();
MyClass* pMyClassInst = new (Align(16)) MyClass();

// and deletion will be straight forward… The pointer passed to
// 'delete' is the same pointer returned from the call to 'new'.
delete pMyClassInst.
```

## Divide and Specialize

A good way to organize memory is to split the memory chunks into small blocks managed by different allocators using various allocation algorithms. This will not just help to be more efficient about the memory usage, but will also serve as a way to clearly assign the memory to the various systems.

### Persistent Allocations

Allocated once at startup/creation of a system. It will never be deleted and therefore we don't need any complicated algorithms to allocate persistent memory. Linear allocators work great in this scenario.

```
// Simple class to handle linear allocations
class LinearAllocator
{
public:
    LinearAllocator(char* pBuffer, int bufferSize) :
        m_pBuffer(pBuffer),
        m_buffersize(bufferSize),
        m_currentOffset(0)
    {
    }
    void* Alloc(size_t size)
    {
       void* pMemToReturn = m_pBuffer + m_currentOffset;
       m_currentOffset += size;
       return pMemToReturn;
    }
    void Free(void* pMem)
    {
       // We can't easily free memory in this type of allocator.
       // Therefore we just ignore this... or you could assert.
    }
private:
    char* m_pBuffer;
    int m_bufferSize;
    int m_currentOffset;
};
```

### Dynamic Allocations

This memory is allocated and freed at random points throughout the lifetime of the application. Sometimes you just need memory to create an instance of some object and you can't predict when that will happen. This is when you can use a dynamic memory allocator. This allocator will have some type of algorithm to remember what memory blocks are free and which ones are allocated. It will handle consolidation of neighboring free blocks. It will however suffer from fragmentation which can greatly reduce the amount of memory available to you. There are tons of dynamic allocation algorithms out there, all with different characteristics; speed, overhead and more. Pick a simple one to start with... you can always change later.

### One-Frame Allocations

These allocations happen for example when one system creates data and another consumes it later in that same frame. It could be variable sized arrays used for rendering, animation or just debug text created by some system earlier in the frame. This allocator handles this by resetting itself at the beginning (or end) of every frame, hence freeing all memory automatically. By doing this we also avoid fragmentation of the memory. The above LinearAllocator can be used here as well with the addition of a simple 'Reset()' function.

## Reference