boost::intrusive_ptr for Object Pooling

Why Object Pooling?

- We can preallocate the memory we are going to use
  - This means less (hopefully no) dynamic allocation in the hot path
  - Memory reuse *might* keep your cache hot

Do we want to use smart pointers?

- Smart pointers are convenient
- They typically come with some overhead
- You may not need/want them at all
- Can be hard (or impossible) to control where delete occurs
  - Can creep into your hot path if you aren't careful

Trying std::shared_ptr

```
#include <atomic>
#include <boost/lockfree/stack.hpp>
#include <cstdint>
#include <memory>

template < typename _T >
class shared_pool
{
private:
   class deleter
   {
   private:
      shared_pool *m_pool;

   public:
      inline deleter(shared_pool *_pool) : m_pool(_pool)
      {
      }

      inline void operator()(_T *_obj)
      {
         m_pool->release(_obj);
      }
   };

private:
   boost::lockfree::stack< int32_t > m_freeObjects;
   _T *m_objects;
   int32_t m_maxSize;
   deleter m_deleter;

public:
   inline shared_pool(int32_t _size)
      : m_freeObjects(_size), m_maxSize(_size), m_deleter(this)
   {
      m_objects = new _T[_size];

      for (int32_t i = 0; i < m_maxSize; ++i)
      {
         m_freeObjects.push(i);
      }
   }

inline ~shared_pool()
   {
      delete[] m_objects;
   }

   inline std::shared_ptr< _T > acquire()
   {
      int32_t index;

      if (m_freeObjects.pop(index))
      {
         return std::shared_ptr< _T >(&m_objects[index], m_deleter);
      }
      else
      {
         return std::shared_ptr< _T >(new _T());
      }
   }

   inline void release(_T *_obj)
   {
      int32_t index = _obj - m_objects;

      if (index >= 0 && index < m_maxSize)
      {
         m_freeObjects.push(index);
      }
      else
      {
         delete _obj;
      }
   }
};
```

std::shared_ptr Pros and Cons

- Pros
  - Easy to implement
  - Supports any type you throw at it
  - Is a very commonly used structure in many code bases
  - Easily made to automatically return objects to the pool when they are no longer referenced
  - Can be carried around in a std::shared_ptr< void > container
- Cons
  - Dynamically allocates the reference counter every time you acquire an object from the pool
    - **This defeats one of the important reasons to pool objects in the first place!**

What Is boost::intrusive_ptr?

- boost::intrusive_ptr is a smart pointer that uses a reference counter **inside** the object that it points to

Trying boost::intrusive_ptr

```
#include <boost/intrusive_ptr.hpp>
#include <atomic>
#include <boost/lockfree/stack.hpp>
#include <cstdint>

template < typename _T >
class intrusive_pool;

template <typename _T >
class intrusive_poolable;

template< typename _T >
class intrusive_poolable
{
friend class intrusive_pool< _T >;

protected:
   std::atomic< std::size_t > m_refCount;
   intrusive_pool< _T > *m_pool;

public:
   inline intrusive_poolable() : m_refCount(0), m_pool(nullptr) {}

private:
   inline void set_pool(intrusive_pool< _T > *_pool) { m_pool = _pool; }

public:

   friend inline void intrusive_ptr_add_ref(intrusive_poolable *_obj)
   {
      ++_obj->m_refCount;
   }

   friend inline void intrusive_ptr_release(intrusive_poolable *_obj)
   {
      if (--_obj->m_refCount == 0)
      {
         _obj->m_pool->release(static_cast< _T * >(_obj));
      }
   }
};
```

Trying boost::intrusive_ptr

```
template < typename _T >
class intrusive_pool
{
private:
   boost::lockfree::stack< int32_t > m_freeObjects;
   _T *m_objects;
   int32_t m_maxSize;

public:
   inline intrusive_pool(int32_t _size)
      : m_freeObjects(_size), m_maxSize(_size)
   {
      m_objects = new _T[_size];

      for (int32_t i = 0; i < m_maxSize; ++i)
      {
         m_objects[i].set_pool(this);
         m_freeObjects.push(i);
      }
   }

   inline ~intrusive_pool()
   {
      delete[] m_objects;
   }

   inline boost::intrusive_ptr< _T > acquire()
   {
      int32_t index;

      if (m_freeObjects.pop(index))
      {
         return boost::intrusive_ptr< _T >(&m_objects[index]);
      }
      else
      {
         _T *obj = new _T();
         obj->set_pool(this);
         return boost::intrusive_ptr< _T >(obj);
      }
   }

   inline void release(_T *_obj)
   {
      int32_t index = _obj - m_objects;

      if (index >= 0 && index < m_maxSize)
      {
         m_freeObjects.push(index);
      }
      else
      {
         delete _obj;
      }
   }
};
```

boost::intrusive_ptr Pros and Cons

- Pros
  - No memory allocation at all!
  - Still get the benefit of smart pointers returning objects to the pool for you
- Cons
  - More difficult to implement
  - Only supports types that inherit from intrusive_poolable< _T >
  - Isn’t a commonly used structure
  - Cannot be carried around as an intrusive_ptr< void >

Conclusion

- If you want to pool objects to avoid memory allocation but still want the convenience of having objects automatically returned to the pool, use boost::intrusive_ptr!