```
/* The following code example is taken from the book
 * "The C++ Standard Library - A Tutorial and Reference"
 * by Nicolai M. Josuttis, Addison-Wesley, 1999
 *
 * (C) Copyright Nicolai M. Josuttis 1999.
 * Permission to copy, use, modify, sell and distribute this software
 * is granted provided this copyright notice appears in all copies.
 * This software is provided "as is" without express or implied
 * warranty, and with no claim as to its suitability for any purpose.
 */
#include <limits>
#include <iostream>

namespace MyLib {
   template <class T>
   class MyAlloc {
     public:
       // type definitions
       typedef T        value_type;
       typedef T*       pointer;
       typedef const T* const_pointer;
       typedef T&       reference;
       typedef const T& const_reference;
       typedef std::size_t    size_type;
       typedef std::ptrdiff_t difference_type;

       // rebind allocator to type U
       template <class U>
       struct rebind {
           typedef MyAlloc<U> other;
       };

       // return address of values
       pointer address (reference value) const {
           return &value;
       }
       const_pointer address (const_reference value) const {
           return &value;
       }

       /* constructors and destructor
        * - nothing to do because the allocator has no state
        */
       MyAlloc() throw() {
       }
       MyAlloc(const MyAlloc&) throw() {
       }
       template <class U>
         MyAlloc (const MyAlloc<U>&) throw() {
       }
       ~MyAlloc() throw() {
       }

       // return maximum number of elements that can be allocated
       size_type max_size () const throw() {
           return std::numeric_limits<std::size_t>::max() / sizeof(T);
```

```
        }

        // allocate but don't initialize num elements of type T
        pointer allocate (size_type num, const void* = 0) {
            // print message and allocate memory with global new
            std::cerr << "allocate " << num << " element(s)"
                      << " of size " << sizeof(T) << std::endl;
            pointer ret = (pointer)(::operator new(num*sizeof(T)));
            std::cerr << " allocated at: " << (void*)ret << std::endl;
            return ret;
        }

        // initialize elements of allocated storage p with value value
        void construct (pointer p, const T& value) {
            // initialize memory with placement new
            new((void*)p)T(value);
        }

        // destroy elements of initialized storage p
        void destroy (pointer p) {
            // destroy objects by calling their destructor
            p->~T();
        }

        // deallocate storage p of deleted elements
        void deallocate (pointer p, size_type num) {
            // print message and deallocate memory with global delete
            std::cerr << "deallocate " << num << " element(s)"
                      << " of size " << sizeof(T)
                      << " at: " << (void*)p << std::endl;
            ::operator delete((void*)p);
        }
    };

    // return that all specializations of this allocator are interchangeable
    template <class T1, class T2>
    bool operator== (const MyAlloc<T1>&,
                     const MyAlloc<T2>&) throw() {
        return true;
    }
    template <class T1, class T2>
    bool operator!= (const MyAlloc<T1>&,
                     const MyAlloc<T2>&) throw() {
        return false;
    }
}
```