```
/* The following code example is taken from the book
 * "The C++ Standard Library - A Tutorial and Reference, 2nd Edition"
 * by Nicolai M. Josuttis, Addison-Wesley, 2012
 *
 * (C) Copyright Nicolai M. Josuttis 2012.
 * 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 <cstddef>
#include <memory>
#include <limits>

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

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

    // allocate but don't initialize num elements of type T
    T* allocate (std::size_t num, const void* hint = 0) {
        // allocate memory with global new
        return static_cast<T*>(::operator new(num*sizeof(T)));
    }

    // deallocate storage p of deleted elements
    void deallocate (T* p, std::size_t num) {
        // deallocate memory with global delete
        ::operator delete(p);
    }

    // return address of values
    T* address (T& value) const {
        return &value;
    }
    const T* address (const T& value) const {
        return &value;
```

```
    }

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

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

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

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

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