# C++ Standard Allocator, An Introduction and Implementation

**Lai Shiaw San Kent**, 18 Aug 2003



Introduction to the allocator concept, as well as implementing a policy-driven allocator template class

**Download source and demo files - 3.52 Kb**

## Introduction

The STL allocator is one of the most overlooked topic in most C++ teachings. They are rarely used explicitly, either via direct client code, or direct construction of an allocator to be used in a container. The possible, and perhaps only area, where you might even notice the word allocator is when you make use of the STL container classes, and wonder what that last parameter (`Allocator`) actually is.

In this article I will explain the purpose of the allocator, what qualifies as a Standard C++ allocator, how an allocator can be implemented, as well as possible usages and extensions.

## The C++ standard purpose

Described in [Josuttis 1999],

> ....Allocators originally were introduced as part of the STL to handle the nasty problem of different pointer types on PCs (such as near, far, and huge pointers). They now serve as a base for technical solutions that use certain memory models, such as shared memory, garbage collection, and object-oriented databases, without changing the interfaces. However, this use is relatively new and not yet widely adopted ....Allocators represent a special memory model and are an abstraction used to translate the need to use memory into a raw call for memory. They provide an interface to allocate, create, destroy, and deallocate objects. With allocators, containers and algorithms can be parameterized by the way the elements are stored. For example, you could implement allocators that use shared memory or that map the elements to a persistent database...

There is, indeed, very little information in [C++] on allocators. It all boils down to simply two sections: 20.1.5 Allocator requirements [*lib.allocator.requirements*], and 20.4.1 The default allocator [*lib.default.allocator*]. In fact, the most important part one should take note is in fact, 20.1.5.1,

> The library describes a standard set of requirements for allocators, which are objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the containers (`_lib.containers_`) are parameterized in terms of allocators.

The provided `std::allocator` in 20.4.1 is the only predefined and required allocator imposed by [C++] on all C++ compiler implementations.

## The C++ standard definition

The standards require the allocator to define types of pointer to `T`(`pointer`), pointer to constant `T`(`const_pointer`), reference to `T`(`reference`), reference to constant `T`, type of `T` itself (`value_type`), an unsigned integral type that can represent the size of the largest object in the allocation model (`size_type`), as well as a signed integral type that can represent the difference between any two pointers in the allocation model (`difference_type`).

The standards then require a template class rebind member, which should pay heed of the following paragraph from 20.1.5.3

> The template class member rebind in the table above is effectively a template `typedef`: if the name `Allocator` is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`.

In short, given `allocator<T>`, we can simply do `allocator::rebind<U>::other.allocate(1)` to be allocating memory large enough to hold an object `U`. This is the magic required for `std::list` to work properly, since given `std::list<int>(allocator<int>())`, `std::list` actually needs to allocate memory for `Node<int>`, and not `int`. Thus, they need to rebind to `allocator<int>()::rebind<Node<int> >::other` instead.

Next, we have to provide a function to simply return the address of a given object (`address`).

What follows is the heart of the allocator, a function to allocate memory for `n` objects of type `T` but not construct the objects (`allocate(n,u)`, where `u` is a hint for other memory models), as well a function to deallocate `n` objects of type `T` (`deallocate(p, n)`). Objects must be destroyed prior to this call.

As mentioned, `allocate` and `deallocate` are simply low level memory management and do not play a part in object construction and destruction. This would mean that the default usage of the keywords `new` and `delete` would not apply in these functions. As any intermediate C++ programmer should know, the following code.

Hide Copy Code

```
A* a = new A;
delete a;
```

is actually [1]interpreted by the compiler as

Hide Copy Code

```
// assuming new throws std::bad_alloc upon failure
A* a = ::operator new(sizeof(A));
a->A::A();
if ( a != 0 ) {  // a check is necessary for delete
    a->~A();
    ::operator delete(a);
}
```

The purpose of the allocator is to [2]allocate raw memory without construction of objects, as well as simply deallocate memory without the need to destroy them, hence the usage of `operator new` and `operator delete` directly is preferred over the usage of the keywords `new` and `delete`.

Following these are helper functions to do copy construction (`construct(p, t)`) and destroy (`destroy(p)`) an object, as well as getting the largest value that can meaningfully be passed to allocate (`max_size`), copy constructor and default constructor, and the equality checking operators(`==` and `!=`).

## A sample allocator

Below is a definition as well as implementation of an allocator that conforms to the C++ standards.

Hide Shrink Copy Code

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

public :
    //    convert an allocator<T> to allocator<U>
    template<typename U>
    struct rebind {
        typedef Allocator<U> other;
    };

public :
    inline explicit Allocator() {}
    inline ~Allocator() {}
    inline explicit Allocator(Allocator const&) {}
    template<typename U>
    inline explicit Allocator(Allocator<U> const&) {}

    //    address
    inline pointer address(reference r) { return &r; }
    inline const_pointer address(const_reference r) { return &r; }

    //    memory allocation
    inline pointer allocate(size_type cnt,
       typename std::allocator<void>::const_pointer = 0) {
      return reinterpret_cast<pointer>(::operator new(cnt * sizeof (T)));
```

```
    }
    inline void deallocate(pointer p, size_type) {
        ::operator delete(p);
    }

    //    size
    inline size_type max_size() const {
        return std::numeric_limits<size_type>::max() / sizeof(T);
 }

    //    construction/destruction
    inline void construct(pointer p, const T& t) { new(p) T(t); }
    inline void destroy(pointer p) { p->~T(); }

    inline bool operator==(Allocator const&) { return true; }
    inline bool operator!=(Allocator const& a) { return !operator==(a); }
};    //    end of class Allocator
```

This is the basic implementation of an allocator, which should be similar to most `std::allocator` provided by your compiler's vendor. However, to the experienced reader, one could immediately identify a possible error. `&r` could only work if `T` did not provide an overloaded `operator &`, and if `T` should, must return the address of the object. Otherwise, the user cannot have a reliable means of obtaining the address of an object `T`.

## Decomposing allocator into policies and traits[3]

Addressing the above address issue (pun intended), it is only logical that we should allow the creator of `T` to actually provide us with the means of obtaining the right address of object `T`. Following a similar design of STL, we could come up with a `Trait` class to provide such a means. As we identified logical designs that should be lifted into a `Trait` class, we should come up with the following design.

Hide Copy Code

```
template<typename T>
class ObjectTraits {
public :
    //    convert an ObjectTraits<T> to ObjectTraits<U>
    template<typename U>
    struct rebind {
        typedef ObjectTraits<U> other;
    };

public :
    inline explicit ObjectTraits() {}
    inline ~ObjectTraits() {}
    template <typename U>
    inline explicit ObjectTraits(ObjectTraits<U> const&) {}

    //    address
    inline T* address(T& r) { return &r; }
    inline T const* address(T const& r) { return &r; }

    inline void construct(T* p, const T& t) { new(p) T(t); }
    inline void destroy(T* p) { p->~T(); }
};    //    end of class ObjectTraits
```

As noted, we have `construct`/`destroy` in as well, since the way the object are constructed/destroyed should be defined by the trait of object `T`. However, take note that `construct`/`destroy` are not always called upon after memory allocation by the containers, nor before memory deallocation by the containers. This is because the containers do not construct objects, they construct copies of objects[2]. Thus, `construct`/`destroy` will not be a reliable means to check for object construction/destruction.

With the `ObjectTraits`, should a creator of `T` decide to `construct`, `destroy`, or overload `operator &`, he could do a complete template specialization of `ObjectTraits` for his purpose.

Following traits, we can also abstract the actual memory allocation/deallocation code into a policy itself.

Hide Shrink Copy Code

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

public :
```

```
    //    convert an StandardAllocPolicy<T> to StandardAllocPolicy<U>
    template<typename U>
    struct rebind {
        typedef StandardAllocPolicy<U> other;
    };

public :
    inline explicit StandardAllocPolicy() {}
    inline ~StandardAllocPolicy() {}
    inline explicit StandardAllocPolicy(StandardAllocPolicy const&) {}
    template <typename U>
    inline explicit StandardAllocPolicy(StandardAllocPolicy<U> const&) {}

    //    memory allocation
    inline pointer allocate(size_type cnt,
      typename std::allocator<void>::const_pointer = 0) {
        return reinterpret_cast<pointer>(::operator
                                    new(cnt * sizeof (T)));
    }
    inline void deallocate(pointer p, size_type)
                           { ::operator delete(p); }

    //    size
    inline size_type max_size() const {
        return std::numeric_limits<size_type>::max();
    }
};    //    end of class StandardAllocPolicy

// determines if memory from another
// allocator can be deallocated from this one
template<typename T, typename T2>
inline bool operator==(StandardAllocPolicy<T> const&,
                       StandardAllocPolicy<T2> const&) {
    return true;
}
template<typename T, typename OtherAllocator>
inline bool operator==(StandardAllocPolicy<T> const&,
                                    OtherAllocator const&) {
    return false;
}
```

The allocation policy determines how the memory allocation/deallocation works, the maximum number of objects of type T that can be allocated, as well as the equality checks to determine if other allocators can allocate and deallocate between allocators interchangeably.

With a simple trait and allocation policy completed, we can now build upon an extensible allocator interface.

Hide Shrink Copy Code

```
template<typename T, typename Policy =
  StandardAllocPolicy<T>, typename Traits = ObjectTraits<T> >
class Allocator : public Policy, public Traits {
private :
    typedef Policy AllocationPolicy;
    typedef Traits TTraits;

public :
    typedef typename AllocationPolicy::size_type size_type;
    typedef typename AllocationPolicy::difference_type difference_type;
    typedef typename AllocationPolicy::pointer pointer;
    typedef typename AllocationPolicy::const_pointer const_pointer;
    typedef typename AllocationPolicy::reference reference;
    typedef typename AllocationPolicy::const_reference const_reference;
    typedef typename AllocationPolicy::value_type value_type;

public :
    template<typename U>
    struct rebind {
        typedef Allocator<U, typename AllocationPolicy::rebind<U>::other,
            typename TTraits::rebind<U>::other > other;
    };

public :
    inline explicit Allocator() {}
    inline ~Allocator() {}
    inline Allocator(Allocator const& rhs):Traits(rhs), Policy(rhs) {}
    template <typename U>
    inline Allocator(Allocator<U> const&) {}
    template <typename U, typename P, typename T2>
    inline Allocator(Allocator<U, P,
       T2> const& rhs):Traits(rhs), Policy(rhs) {}
};    //    end of class Allocator
```

```
// determines if memory from another
// allocator can be deallocated from this one
template<typename T, typename P, typename Tr>
inline bool operator==(Allocator<T, P,
   Tr> const& lhs, Allocator<T,
   P, Tr> const& rhs) {
    return operator==(static_cast<P&>(lhs),
                      static_cast<P&>(rhs));
}
template<typename T, typename P, typename Tr,
        typename T2, typename P2, typename Tr2>
inline bool operator==(Allocator<T, P,
    Tr> const& lhs, Allocator<T2, P2, Tr2> const& rhs) {
      return operator==(static_cast<P&>(lhs),
                      static_cast<P2&>(rhs));
}
template<typename T, typename P, typename Tr, typename OtherAllocator>
inline bool operator==(Allocator<T, P,
          Tr> const& lhs, OtherAllocator const& rhs) {
    return operator==(static_cast<P&>(lhs), rhs);
}
template<typename T, typename P, typename Tr>
inline bool operator!=(Allocator<T, P, Tr> const& lhs,
                       Allocator<T, P, Tr> const& rhs) {
    return !operator==(lhs, rhs);
}
template<typename T, typename P, typename Tr,
           typename T2, typename P2, typename Tr2>
inline bool operator!=(Allocator<T, P, Tr> const& lhs,
                  Allocator<T2, P2, Tr2> const& rhs) {
    return !operator==(lhs, rhs);
}
template<typename T, typename P, typename Tr,
                             typename OtherAllocator>
inline bool operator!=(Allocator<T, P,
        Tr> const& lhs, OtherAllocator const& rhs) {
    return !operator==(lhs, rhs);
}
```

Notice the usage of public inheritance of policy and traits, as opposed to having them as a member data. This enables each allocator instance to have his own memory management model (via the policy), as well as take advantage of EBCO (Empty Base Class Optimization) if available by the compiler, since in most cases, traits would be an empty class.

The `Allocator` class usage is as simple as follows,

Hide Copy Code

```
std::vector<int, Allocator<int> > v;
```

## Memory allocation tracking policy implementation

The previous listed allocator does the most basic memory management. Building on a working model, we could actually perform profiling on memory management, for example, by having the following memory allocation tracking policy,

Hide Shrink Copy Code

```
template<typename T, typename Policy = StandardAllocPolicy<T> >
class TrackAllocPolicy : public Policy {
private :
    typedef Policy AllocationPolicy;

public :
    //    convert an TrackAllocPolicy<T> to TrackAllocPolicy<U>
    template<typename U>
    struct rebind {
        typedef TrackAllocPolicy<U,
           typename AllocationPolicy::rebind<U>::other> other;
    };

public :
    inline explicit TrackAllocPolicy():total_(0), current_(0), peak_(0) {}
    inline ~TrackAllocPolicy() {}
    inline explicit
        TrackAllocPolicy(TrackAllocPolicy const& rhs):Policy(rhs),
        total_(rhs.total_), current_(rhs.current_), peak_(rhs.peak_) {}
    template <typename U>
    inline explicit
       TrackAllocPolicy(TrackAllocPolicy<U> const& rhs):Policy(rhs),
       total_(0), current_(0), peak_(0) {}
```

```
    //    memory allocation
    typename AllocationPolicy::pointer
      allocate(typename AllocationPolicy::size_type cnt,
      typename std::allocator<void>::const_pointer hint = 0) {
        typename AllocationPolicy::pointer p =
               AllocationPolicy::allocate(cnt, hint);
        this->total_ += cnt;
        this->current_ += cnt;
        if ( this->current_ > this->peak_ ) {
            this->peak_ = this->current_;
        }
        return p;
    }
    inline void deallocate(typename AllocationPolicy::pointer p,
        typename AllocationPolicy::size_type cnt) {
        AllocationPolicy::deallocate(p, cnt);
        this->current_ -= cnt;
    }

    // get stats
    inline typename AllocationPolicy::size_type
          TotalAllocations() { return this->total_; }
    inline typename AllocationPolicy::size_type
          CurrentAllocations() { return this->current_; }
    inline typename AllocationPolicy::size_type
          PeakAllocations() { return this->peak_; }

private :
    //    total allocations
    typename AllocationPolicy::size_type total_;
    //    current allocations
    typename AllocationPolicy::size_type current_;
    //    peak allocations
    typename AllocationPolicy::size_type peak_;
};    //    end of class TrackAllocPolicy

// determines if memory from another
// allocator can be deallocated from this one
template<typename T, typename Policy, typename T2, typename Policy2>
inline bool operator==(TrackAllocPolicy<T, Policy> const& lhs,
        TrackAllocPolicy<T2, Policy2> const& rhs) {
  return operator==(static_cast<Policy&>(lhs),
                    static_cast<Policy&>(rhs));
}
template<typename T, typename Policy, typename OtherAllocator>
inline bool operator==(TrackAllocPolicy<T,
     Policy> const& lhs, OtherAllocator const& rhs) {
  return operator==(static_cast<Policy&>(lhs), rhs);
}
```

This allocation policy merely adds tracking capability, and is built upon another/actual memory allocation policy, determined by the second template argument.

The class usage is as simple as follows,

Hide Copy Code

```
std::vector<int, Allocator<int, TrackAllocPolicy<int> > > v;
```

## Small Object Allocator

Another possible implementation of an allocation policy might be optimization for small objects allocation. Frequent allocation/deallocation of small objects from the free store can, at times, hurt the performance of an application. A work-around for this is to allocate a block of large memory at one time, and hand out these memory to the application upon request. Deallocated small objects are returned to the block, and reused at a later date.

Since Loki[5] has a Small Object Allocator class done, it would be logical for us to adapt it as our allocator.

Hide Shrink Copy Code

```
template<typename T, std::size_t numBlocks = 64>
class SmallObjectAllocPolicy {
public :
    //    typedefs
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
```

```
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

public :
    //    convert an SmallObjectAllocPolicy<T> to SmallObjectAllocPolicy<U>
    template<typename U>
    struct rebind {
        typedef SmallObjectAllocPolicy<U, numBlocks> other;
    };

public :
    inline explicit SmallObjectAllocPolicy() {}
    inline ~SmallObjectAllocPolicy() {}
    inline explicit SmallObjectAllocPolicy(SmallObjectAllocPolicy const&) {}
    template <typename T2, std::size_t N2>
    inline explicit
      SmallObjectAllocPolicy(SmallObjectAllocPolicy<T2, N2> const&) {}

    //    memory allocation
    inline pointer allocate(size_type cnt,
          typename std::allocator<void>::const_pointer = 0) {
        return reinterpret_cast<T*>(allocator_.Allocate(sizeof(T) * cnt));
    }
    inline void deallocate(pointer p, size_type cnt) {
        allocator_.Deallocate(p, sizeof(T) * cnt);
    }

    //    size
    inline size_type max_size() const {
        return std::numeric_limits<size_type>::max() / sizeof(T);
    }

private:
    static Loki::SmallObjAllocator allocator_;
};    //    end of class SmallObjectAllocPolicy

//    optimized for single small object,
//    hence chunk size and max object size is small
//    otherwise using free store
template<typename T, std::size_t numBlocks>
Loki::SmallObjAllocator SmallObjectAllocPolicy<T,
         numBlocks>::allocator_(numBlocks * sizeof(T), sizeof(T));

//    determines if memory from another allocator
//    can be deallocated from this one
template<typename T, std::size_t N>
inline bool operator==(SmallObjectAllocPolicy<T, N> const&,
        SmallObjectAllocPolicy<T, N> const&) {
    return true;
}
template<typename T, std::size_t N, typename T2, std::size_t N2>
inline bool operator==(SmallObjectAllocPolicy<T,
       N> const&, SmallObjectAllocPolicy<T2, N2> const&) {
    return false;
}
template<typename T, std::size_t N, typename OtherAllocator>
inline bool operator==(SmallObjectAllocPolicy<T,
            N> const&, OtherAllocator const&) {
    return false;
}
```

If you would like to understand how Loki's `SmallObjAllocator` works, remember to check out the code provided with Loki, or purchase [Alexandrescu 2001].

Using the `SmallObjectAllocator` is easy as well.

Hide Copy Code

```
std::list<int, Allocator<int, SmallObjectAllocPolicy<int, 64> > > vTest;
```

If you had looked into the Loki's `SmallObjAllocator`, you would note that it works best for single allocation of a fixed size (for allocation of different sizes, they actually create different internal allocators). Thus, it would not work well with `std::vector` or other similar containers that does allocation of a block of memory (and of varied size).

Of course, if before hand, you knew the maximum size of your structure, you could do the following as well,

Hide Copy Code

```
std::vector<int> v;
v.reserve(10);
```

Alternatively, if you want to do away with dynamic memory allocation, you could declare an allocator that provides a fixed-sized array,

though with the way `std::vector` takes in an allocator as its constructor, you would end up with an unnecessary temporary. (Nevertheless, the codes are provided in the source, and it only works if and only if you know the size before hand, and have used the `reserve` function)

## Resolving multi-threading issues

Initially I had wanted to have custom threading policies in the `Allocator` class, because, for example, there's `SmallObjectAllocator`, which used a shared memory pool for giving out objects `T`. Multiple thread using `SmallObjectAllocator<T, N>` might actually cause inconsistent state of the keeping track of internal memory pool. Except that a mutex/lock of any form didn't work out in the `Allocator` (Thanks to Sean Kent for pointing that out), and even in `SmallObjectAllocator<T, N>,` since it is still different instances, but sharing the same static `Loki::SmallObjAllocator`. The only possible solution would be to actually perform the lock/mutex within a wrapper for `Loki::SmallObjAllocator`, or even `Loki::SmallObjAllocator` itself.

Thus, this implementation is actually not thread-safe by itself (at least in the case of `SmallObjectAllocator`).

## Conclusion

The allocator concept is a very powerful method to encapsulate your memory management model, and it does not touch on the overloading operator `new` and `delete`, which some see it as bad form or evil. With allocators, you can plug memory tracking, optimized memory management model for certain objects, etc into your containers. If you designed and built your framework around allocators, the possibilities are immerse.

Of course, do not go into the "Golden hammer" syndrome, or what some would describe as "When you have a hammer, everything else looks like a nail". There are certain designs where plugging in an allocator would not be logical, and although not much harm can be done (other than perhaps build times and dependency), it offers not much benefit as well.

## Footnotes

[1] Described in [Lippman 1996], Chp 5, Semantics of Construction, and Copy.

[2] In STL Containers, there is no need for construction of objects via the default constructor, because the data in the container are merely a copy created from the actual data inserted (using the copy constructor).

[3] Credits and recognition should be given to Sean Kent for the original idea and implementation of the policy-based allocator. Note that there are similarities and differences between the version described here and his original version.

[4] Refer to Template Specialisation - An Introductory Survey By Henrik Stuart for an excellent introduction and explanation on template specialization.

[5] Remember to download the Loki library.

## References


- [Josuttis 1999]: Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference* Addison-Wesley Pub Co 1999
- [C++]: C++ Standards, 1998
- [Lippman 1996]: Stanley B. Lippman, *Inside the C++ Object Model* Addison-Wesley Pub Co, 1996
- [Alexandrescu 2001]: Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Pub Co, 2001


## History

- 19th August 2003: Initial version uploaded.

## License



## Share

## About the Author


**Lai Shiaw San Kent**
Web Developer
Singapore

No Biography provided

## You may also be interested in...



## Comments and Discussions

However, the downloaded source files refer to "missing" header files.

I strongly feel that provided source should BUILD.

In the case of an example allocator to be used in conjunction with the standard library, I think the example should be OS-independent.

IMO, a downloadable project should either

A) Contain all source files needed to build (compile and link), other than those generally provided with a compiler.

B) Provide a link or information about where to obtain any needed "third party" source files.

C) If the provided source is OS-dependent, specify the requirements.

Sign In · View Thread · Permalink

**Re: Nice if it built...** Member 10643799 16-Jun-14 10:19

**allocate method question** **Michael HS** **3-Aug-11 9:23**

Hello,

I'm busy looking into the stuff of C++ that I haven't really used before. Allocators are part of it and this article already helped a great deal.
There is however one thing I'm confused about and that's the prototype of the allocate function:

Hide Copy Code

```
inline pointer allocate(size_type cnt,
      typename std::allocator<void>::const_pointer = 0) {
        return reinterpret_cast<pointer>(::operator
                                    new(cnt * sizeof (T)));
    }
```

Could you explain the function of the second parameter "typename std::allocator::const_pointer = 0", why should it be there and what does it do? As I don't see it being used in the function at all...

Anyway, I just don't get it and any input on this is welcome,

Thanks

Sign In · View Thread · Permalink

**Re: allocate method question** Michael HS 3-Aug-11 21:47

**Nitpick** **nick.strupat** **31-Oct-10 9:55**

I think the member 'address' that returns a const_pointer should be a const member. What do you think?

Sign In · View Thread · Permalink

**Constructor orders flipped in class Allocator** **Rhys Ulerich** **17-Sep-10 10:26**

A minor nit. Within
`class Allocator : public Policy, public Traits { /* ... */ }`
the two copy constructors have `: Traits(rhs), Policy(rhs)` instead of the reverse. It triggers a GCC warning.

Sign In · View Thread · Permalink

**Does boost::addressof alleviate the need for ObjectTraits?** **Rhys Ulerich** **18-Mar-10 11:08**

It looks like boost::addressof eliminates the problems that ObjectTraits::address was intended to solve. Would it make sense to use boost::addressof within ObjectTraits::address to almost always eliminate the need to specialize ObjectTraits for classes where operator& is overriden?

http://www.boost.org/doc/libs/1_42_0/libs/utility/utility.htm#addressof[^]

Sign In · View Thread · Permalink

**Problems in operator==** — Rhys Ulerich — 15-Mar-10 6:46

I believe the operator== has some issues with the static_cast's and should resemble

Hide Copy Code

```
template<typename T, typename P, typename Tr>
inline bool operator==(allocator<T, P, Tr> const& lhs,
                       allocator<T, P, Tr> const& rhs)
{
    return operator==(static_cast<P const&>(lhs), static_cast<P const&>(rhs));
}
```

where I've added `const` to the `static_cast`s

Sign In · View Thread · Permalink 5.00/5 (1 vote)

**specialization for void?** — Rhys Ulerich — 14-Mar-10 12:54

Does Allocator require any template specialization for Allocator? Been reading around, and it seems like some special handling may be required (e.g. )[^].

Sign In · View Thread · Permalink

**Doesn't work with Loki 0.1.5** — tamboril2 — 20-Sep-06 4:39

I'm getting lots of errors with Loki 0.1.5 when trying to use the SmallObjAllocator.

Sign In · View Thread · Permalink

**Look my CustomNewDeleteAllocPolicy** — WRice — 15-Sep-05 21:57

This is my code.
How do you think about this?

You can use this with Loki's SmallObject class.
Like this..

Hide Copy Code

```
CustomNewDeleteAllocPolicy< T, SmallObject<> >
```

You can use custom allocator class

Hide Copy Code

```
class MyMemoryAllocator
{
    static void * operator new(std::size_t size) { return MyMemMan::Malloc(size); }
    static void operator delete(void * p, std::size_t size) { MyMemMan::Free(p, size); }
}
CustomNewDeleteAllocPolicy< T, MyMemoryAllocator >
```

Hide Expand Copy Code

```
///////////////////////////////////////////////////////////////////////////
//  class CustomNewDeleteAllocPolicy
///////////////////////////////////////////////////////////////////////////

    //  an allocator use over-rided new/delete method
    //  you can use Loki's SmallObject. CustomNewDeleteAllocPolicy< T, SmallObject<> >
    template<typename T, typename CustomNewDelete>
    class CustomNewDeleteAllocPolicy {
    public :
        //  typedefs
        typedef T value_type;
        typedef value_type* pointer;
        typedef const value_type* const_pointer;
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef std::size_t size_type;
        typedef std::ptrdiff_t difference_type;

    public :
        //  convert an CustomNewDeleteAllocPolicy<T> to CustomNewDeleteAllocPolicy<U>
        template<typename U>
```

```
        struct rebind {
            typedef CustomNewDeleteAllocPolicy<U, CustomNewDelete> other;
        };

    public :
        inline explicit CustomNewDeleteAllocPolicy() {}
```

http://www.wrice.org
WRice means White Rice.

Sign In · View Thread · Permalink

**TackAllocPolicy is Not working well.** **Kwak Jae Hyuk** **13-Sep-05 3:56**

I using this code.
But, TrackAllocPolicy is not working well.
My code is here.

typedef vector< int, Allocator< int, TrackAllocPolicy< int, SmallObjectAllocPolicy< int > > > > MyVectorType;

MyVectorType k1;
k1.push_back(30);

MyVectorType k2(k1); // problem!!
size_t Cur2 = k2.get_allocator().CurrentAllocations(); // return 2!!!! I expected 1
size_t Cur1 = k1.get_allocator().CurrentAllocations(); // return 1

MyVectorType k3;
k3 = k1;
size_t Cur3 = k3.get_allocator().CurrentAllocations(); // return 1

Plz, check is problem.

White Rice - http://www.wrice.org

Sign In · View Thread · Permalink

**Good article** **Christopher Diggins** **17-Jun-05 8:25**

Good article. One small point, all of those inlines are superflous. When functions are defined inside of the class they are supposed to be automatically treated as if declared inline. (At least that is what Stroustrup says in TCPL 3).

Keep up the good work.

Christopher Diggins - Object Oriented Template Library ( OOTL )

Sign In · View Thread · Permalink 2.00/5 (1 vote)

**Re: Good article** dgomezpr 3-Aug-07 6:32

Last Visit: 31-Dec-99 18:00 Last Update: 7-Aug-16 16:16 Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin