Splitting *node* and *array* allocation in allocators

Document number: P0310R0
Date: 2016-03-19
Project: Programming Language C++
Audience: Library Evolution Working Group
Reply to: Marcelo Zimbres (mzimbres@gmail.com)

Abstract: This is a non-breaking proposal to the C++ standard that aims to reduce allocator complexity, support realtime allocation and improve performance of node-based containers by making a clear distinction between *node* and *array* allocation in the `std::allocator_traits` interface. Two new member functions are proposed, `allocate_node` and `deallocate_node`. We also propose that the container node type should be exposed to the user. A prototype implementation is provided.

Size management adds undue difficulties
and inefficiencies to any allocator design
A. ALEXANDRESCU

Contents

1 Introduction

THE IMPORTANCE of linked data structures in computer science, like trees and linked lists, cannot be over-emphasised, yet, in the last couple of years it has become a common trend in C++ to move away from such data structures due to their sub-optimal memory access patterns [1, 2, 3]. In fact, many people today prefer to use the flat alternatives and pay $O(n)$ insertion time, than $O(1)$ at the cost of memory fragmentation and unpredictable performance loss.

We believe in fact, that the "*Don't pay for what you don't use*" premise is not being met on node-based containers due to the restrictive array-oriented allocator interface. This proposal tries to fix what the author believes to be the root of problem: *The lack of distinction between node and array allocation.* We propose here a complete split between these two allocation techniques by means of a non-breaking addition to `std::allocator_traits`.

1.1 Node allocation

Node allocation is one of the simplest allocation techniques available, but yet a very powerful one. The simplicity comes from the fact that allocation and deallocation reduces to pushing and popping from the linked list of nodes, like shown in the code below. It is powerful because it is very fast, perform in hard-real-time and causes minimal memory fragmentation.

```
1  pointer allocate(std::size_t /*can't handle n*/)
2  {
3    pointer q = avail; // The next free node
4    if (avail)
5      avail = avail->next;
6
7    return q;
8  }
9
10 void deallocate(pointer p, std::size_t /*can't handle n*/)
11 {
12   p->next = avail;
13   avail = p;
14 }
```

The reasons why this allocation technique is not fully supported in C++ is related to the current *array* oriented interface of allocators. The member function `allocate(n)` may be called with $n \neq 1$, meaning that the alloca-

tor now has to implement array allocation strategies instead of much more simple and efficient node allocation.

Before going into more details, let us see with a use case, how this proposal provides better solution over the current array oriented interface.

1.2 Use case

The example below uses the allocation technique from the previous section to write code that is fast, simple and uses the minimum amount of memory. A linked list served with a couple of nodes allocated on the stack

```
1  using alloc_t = rt::node_allocator<int>;
2  using node_type = typename std::list<int, alloc_t>::node_type;
3
4  // Buffer for 100 elements.
5  std::array<char, 100 * sizeof (node_type)> buffer = {{}};
6  alloc_t alloc(buffer);
7
8  std::list<int, alloc_t> l1(alloc);
9  // Inserts elements. Allocation and deallocation implemented
10 // with 6 lines of code.
11 l1 = {27, 1, 60};
12 ...
```

Some of the features of this code are

- It uses the container node type, to calculate the minimum amount of memory to support 100 elements in the list. As a consequence the buffer is compact, improving cache locality and causing minimal fragmentation.
- The allocator knows it is doing node allocation and does not make the node size bigger to store bookkeeping information. *You do not pay for space you do not use.*
- Very simple and fast allocator where allocation and deallocation translates into only a couple of pointers assignments. No array allocation strategy had to be implemented.

The reasons why we cannot write this code in current C++ will be better explained below, but shortly said

- The allocator *has to* provide array allocation since `allocate(n)` may be called with $n \neq 1$, as a result the allocator gets unnecessarily complicated and the size of the buffer to support 100 elements is not anymore clear since it depends on the array allocation strategy/algorithm.
- The container node type and therefore its size is unknown.

1.3 Exposing the node type

In current C++, there is no straightforward way of knowing the size of the node the allocator will serve. At runtime it is known only when the rebound allocator instance is constructed, which occurs when the container is constructed. It is a tricky to use this information. As shown in the example above the user may want to use it to pre-allocate space for a certain number of elements.

Another situation where the node type is useful is when implementing node allocators for unordered containers. Usually, unordered containers rebind twice and there is no way of knowing which rebound type is used for array or node allocation. Once the node type is exposed the allocator can be specialized for the desired type, offering node allocation functions accordingly.

The difficult in exposing the node type is the recursiveness of the problem. The node type is not known until the container type is known, which in turn depends on the allocator type to be defined and the allocator type cannot be defined before the node type is known.

At a first glance we may quite naturally demand the node type to be independent of the container and of the allocator, however, due to the support for fancy pointers in c++, the following cannot be implemented in general

```
1 // Cannot always hold for general A1 nd A2.
2 static_assert( std::is_same< std::list<T, A1>::node_type
3                             , std::list<T, A2>::node_type>, "");
```

The solution we propose here is to offer a rebind structure in the node type, so that the specialization can get rid of the allocator pointer type, that the node type happened to be defined with. In other words, we can get the node type from a container defined with any allocator and rebind to a node with a different pointer type. Recursiveness is bypassed this way.

1.4 Further considerations

The influence of fragmentation on performance is well known on the C++ community and subject of many talks in conferences, therefore I am not going to repeat results here. The interested reader can refer to [2, 3] for example.

The split between node and array allocation has been successfully implemented in the Boost.Container library, but the mechanism is based on C++03 instead of `std::allocator_traits`.

For an allocator that explores features proposed here, please see the project [6]. For a general talk on allocators and why size management is a problem [7]. For related proposal, please see [5]. For an alternative approaches to support node allocation, please see appendix A.

2 Motivation and scope

Given the popularity of the standard allocator, it is important to give reasons why it should be avoided as a first option for node allocations. I will focus on the use case given above, where we want to serve an `std::list<int>` with a certain number of nodes.

2.1 Why avoid the standard allocator

1. Nodes go necessarily on the heap. For only 100 elements I would preferably use the stack.

2. The node size is small ($\approx$ 20bytes), it is not recommended allocating them individually on the heap. Fragmentation begins to play a role if I have many lists or bigger n (see benchmarks below).

3. Each heap allocation is an overhead: all the code inside malloc, plus system calls and allocation strategies. (I only need 20 bytes of space for a node!). Most importantly, the standard allocator does not know we are doing node allocations and cannot optimize it.

4. Unknown allocated size. Does it allocate more space to store information needed by the algorithm? How much memory I am really using?

All this is overkill for a simple list with a couple of elements. When the number of elements gets bigger and the nodes go to the heap, the situation gets much worser for standard allocator or any custom allocator that implements array allocation strategies. This is the topic of the next section.

2.2 Benchmarks

In the previous section we gave some motivation on why one should avoid the standard allocator, but what about a custom allocator? To test how much improvement we can get with custom allocators I tested my own non-optimized implementation of a node allocator against allocators shipped with GCC. The graphs can be seem below. The node allocator has never been slower, in fact, it was most of the time faster than any other fine-tuned allocator.

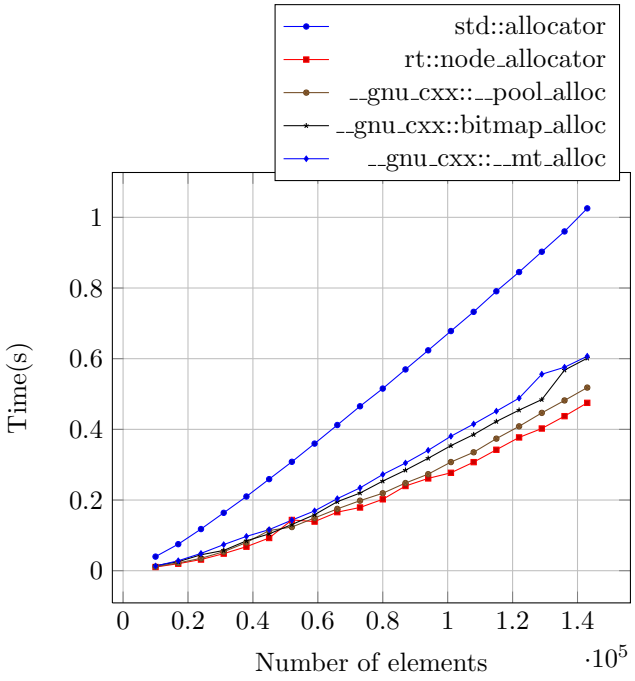

Figure 1: Source code can be found in [6].

2.3 General motivations

Let us see some general motivations on why support for node allocation is desirable

1. Support the most natural and one of the fastest allocation scheme for linked data structures. In `libstd++` and `libc++` for example, it is already possible (by chance) to use this allocation technique, since n is always 1 on calls of `allocate(n)`.

2. Node-based containers do not manage allocation sizes but unnecessarily demand this feature from their allocators, with a cost in performance and overall allocator complexity.

3. Support hard-realtime allocation for node-based containers through pre-allocation and pre-linking of nodes. This is highly desirable to improve C++ usability in embedded systems.

4. State of the art allocators like `boost::node_allocator` [4] achieve great performance gains optimizing for the $n = 1$ case.

5. Avoid wasted space behind allocations. It is pretty common that allocators allocate more memory than requested to store information like the size of the allocated block.

6. Keep nodes in as-compact-as-possible buffers, either on the stack or on the heap, improving cache locality, performance and making them specially useful for embedded programming.

3 Impact on the Standard

THE FOLLOWING additions are required in the standard.

3.1 New `std::allocator_traits` members

We require the addition of two new member functions and a typedef in `std::allocator_traits` as follows

```
template<class Alloc>
struct allocator_traits {
  // Equal to Alloc::node_allocation_only if present,
  // std::false_type otherwise. Array allocation with
  // allocate(n) is ruled out if it is std::true_type.
  using node_allocation_only = std::false_type
  // Calls a.allocate_node() if present otherwise calls
  // Alloc::allocate(1). Memory allocate with this function
  // must be deallocated with deallocate_node.
  pointer allocate_node(Alloc& a);
  // Calls a.deallocate_node(pointer) if present otherwise
  // calls Alloc::deallocate(p, 1). Can only be used with
  // memory allocated with allocate_node.
  void deallocate_node(Alloc& a, pointer p);
};
```

These additions provide the following options inside node-based containers

1. **Array allocation only**. This is the *status quo*. Libraries can continue to call `allocate(n)` if they want, but since the majority of implementations use $n = 1$, they may be simply implemented with `allocate_node()`, regardless of whether the allocator provides this function or not. The implementation of `allocate_node()` in the `std::allocator_traits` falls back to `allocate(1)` when the allocator does not provide one.

2. **Node allocation only**. In this case, the user is required to set the typedef `node_allocation_only` to `std::true_type` in the allocator and provide `allocate_node()`. The user is not required to provide `allocate(n)`.

3. **Array and node allocation together**. It is possible to use both array and node allocation when the user provides `allocate_node` and sets `node_allocation_only` to `std::false_type`. I am unaware if this option is useful.

3.2 The node type

We require the following node interface on node based containers

```
template <class T, class Ptr>
struct node_type {
  using value_type = T;
  using pointer = // Usually taken from std::pointer_traits<Ptr>
  template<class U, class K>
  struct rebind { using other = node_type<U , K>; };
  // ... implementation details
};
```

We also require the node type to be independent of the container with the exception of the allocator type, for example, the following code should compile.

```
using set_type1 = rt::set<T, C1, A1>;
using set_type2 = rt::set<T, C2, A2>;

using pointer = // Arbitray pointer type.
using node_type1 =
  typename set_type1::node_type::template rebind<T, pointer>;
using node_type2 =
  typename set_type2::node_type::template rebind<T, pointer>;
static_assert(std::is_same<node_type1, node_type2>::value,"");
```

All node-based containers are affected: `std::forward_list`, `std::list`, `std::set`, `std::multiset`, `std::unordered_set`, `std::unordered_multiset`, `std::map`, `std::multimap`, `std::unordered_map`, `std::unordered_multimap`

4 Acknowledgment

I WOULD like thank people that gave me any kind of feedback: Ville Voutilainen, Nevin Liber, Daniel Gutson, Alisdair Meredith. Special thanks go to Ion Gaztañaga for suggesting important changes in the original design and David Krauss for suggesting other approaches.

5 References

[1] Sean Middleditch, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0038r0.html

[2] Chandler Carruth, *Efficiency with Algorithms, Performance with Data Structures* (https://www.youtube.com/watch?v=fHNmRkzxHWs)

[3] Scott Meyers, *Cpu Caches and Why You Care* (https://www.youtube.com/watch?v=WDIkqP4JbkE)

[4] http://www.boost.org/doc/libs/1_58_0/boost/container/node_allocator.hpp

[5] Ion Gaztañaga, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2045.html

[6] https://github.com/mzimbres/rtcpp

[7] Andrei Alexandrescu, *std::allocator Is to Allocation what std::vector Is to Vexation* (https://www.youtube.com/watch?v=LIb3L4vKZ7U)

[8] http://www.open-std.org/pipermail/embedded/2014-December/000335.html

[9] https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/ccwOpTxM_xE

[10] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2980.pdf

A Alternative approaches

THERE ARE other possible approaches to support node allocation that are worth knowing of. I will describe them here, so that the committee can compare them.

A.1 `allocate(n)` with $n = 1$

This seems the easiest way to perform node allocation. Once the standard guarantees n will be always 1, there is no more need to provide new node allocation functions for node-based containers. The parameter n can be simply ignored. The `allocate` and `deallocate` can be implemented in terms of node-allocation-only functions, for example

```
pointer allocate(std::size_t /* n is ignored */)
{
  return allocate_node();
}

void deallocate(pointer p, std::size_t /* n is ignored */)
{
  deallocate_node();
}
```

The problem with this approach is that it prevents array allocation inside node-based containers, which means it can be viewed as a narrowing of the current interface.

A.2 Provide a `constexpr max_size()` that returns 1

This scheme can achieve the same goals as my main proposal and does not require any addition to `std::allocator_traits`. Libraries should check if

`max_size()` can be evaluated at compile time and take appropriate action i.e. ensure `allocate(n)` is always called with $n = 1$. I did not adopted it due to some disadvantages I see with it

1. It does not make containers implementation simpler.
2. Function names should reflect that array and node allocation have different semantics, apart from the storage size. If memory expansion (realloc) is added in the future, it should only work with storage allocated with `allocate(n)` but not with storage allocated for nodes. This allows node allocations to avoid extra bookkeeping data to mark the storage as non-expandable.
3. It requires the user to specialize `std::allocator_traits` to provide a `constexpr max_size()` since the default is not `constexpr`. This is not bad but I prefer to avoid it if I can.
4. Other static information like `propagate_on_container_copy_assignment`, etc, are provided as typedef so I prefer to keep the harmony.
5. It sounds more like a hack of the current allocator interface to achieve node allocation than a full supported feature.