Improving allocator interface for node-based containers

Document number: 1
Date: February 20, 2016
Project: Programming Language C++
Author: 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.

Contents

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

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 array and node allocation*. We propose here a complete split between these two allocation techniques by means of a non-breaking addition to `std::allocator_traits`.

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

Use case. The example below uses the proposed features to write code that is fast, simple and uses the minimum amount of memory. A linked lists served with a couple of nodes allocated on the stack

```
1  using alloc_t = rt::node_allocator<int>;
2
3  using node_type = typename std::list<int, alloc_t>::node_type;
4
5  // Buffer for 100 elements. offset is a space reserved for
6  // the allocator (three computer words in my implementation).
7  std::size_t node_size = sizeof(node_type);
8  std::array<char, offset + 100 * node_size> buffer = {{}};
9
10 alloc_t alloc(buffer);
11
12 std::list<int, alloc_t> l1(alloc);
13 ...
14 // Inserts elements. Allocation and deallocation implemented
15 // with 6 lines of code (see below). Faster and simpler than
16 // any allocator I could find.
17 l1 = {27, 1, 60};
18 ...
```

Some of the features of this code are

- It uses the container node type, to calculate the amount of memory needed to support 100 elements in the list.
- The buffer is compact with the optimal size, 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 allocator, only some pointers change. (see below)
- *It cannot be written portably in C++.*

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

- The container node type and therefore its size is unknown.
- The function `allocate(n)` may be called with $n \neq 1$, meaning that the allocator now has to implement array allocation strategies instead of much more simple and efficient node allocation.
- The size of the buffer is not anymore clear and depends on the allocation strategy/algorithm.

How simple is node allocation. The simplification of having to provide only node allocation in the allocator comes from the fact that we can simply build a singly linked list where allocation and deallocation reduces to pushig and popping from the linked list

```
1  pointer allocate(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, size_t /*can't handle n*/)
11 {
12   p->next = avail;
13   avail = p;
14 }
```

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 for the sake of readability. The interested reader can refer to [2, 3] for example.

For an allocator that explores the feature 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

Consider the example from last section where a programmer needs an `std::list` to store some (≈ 100) integers somewhere in the code. In current C++, there are essentially two approaches to do this (if we want to use a standard container). We can simply use the standard allocator `std::list<int>`. This is undesirable for many reasons

1. Nodes go 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).

3. Each heap allocation is an overhead: all the code inside malloc, plus system calls and allocation strategies. (I only need 20bytes 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 (for bigger n situation gets worser). At this point we think it is better to use a custom allocator but as we saw in the last secton, we cannot improve too much give the current array-oriented allocator interface.

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 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

We require the addition of two new member function 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.
  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 simply be implemented in terms of `allocate_node()`, regardless of whether the allocator provides this function or not. The implementation of `allocate_node()` in `std::allocator_traits` should fall 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.

Exposing the node type. Even though the changes proposed above are enough to achieve efficient node allocation, we still have no direct means of knowing the allocation size i.e. the node size. This information is not available at compile time and at runtime only when the rebound allocator instance is constructed on container construction. It is a tricky operation that can be avoided if we expose the container node type.

Another situation where the node type is useful is when implementing node allocators for unordered containers. Usually, unordered containers rebound twice and there is no way of knowing which rebound type is used for array and node allocation. Once the node type is exposed it is easy to detect the type that will be used for node allocation for example. We require that the exposed node type support SCARY initialization [10]. The following should hold

```
1 using node_type1 = std::set<int, C1, A1>::node_type;
2 using node_type2 = std::set<int, C2, A1>::node_type;
3 using node_type3 = std::set<int, C1, A2>::node_type;
4
5 static_assert(std::is_same<node_type1, node_type2>::value, "");
6 static_assert(std::is_same<node_type1, node_type3>::value, "");
7 static_assert(std::is_same<node_type2, node_type3>::value, "");
```

The following 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`

Unordered containers. Unordered containers have to rebind twice, where one type is used for node allocation and one for array allocation. Let us name these types `node_alloc_type` and `array_alloc_type`. According to this proposal, `node_alloc_type` is not required to provide the `allocate(n)` member whenever `node_allocation_only` is set to `std::true_type()`.

Implementing such allocators for node based containers is easy when we know the container node type. An example implementation is provided in [6].

4 Acknowledgment

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

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 list them here, so that the commitee can compare them.

Ensure `allocate(n)` is called 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 array allocation 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_node(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.

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 bookeeping 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.