Improving allocator interface for node-based containers

Document number: 1
Date: February 14, 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 containers 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 allocations*. We propose here a complete split between these two allocation schemes by means of a non-breaking addition to `std::allocator_traits`

```
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 be simply 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 with 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

```
using node_type1 = std::set<int, C1, A1>::node_type;
using node_type2 = std::set<int, C2, A1>::node_type;
using node_type3 = std::set<int, C1, A2>::node_type;

static_assert(std::is_same<node_type1, node_type2>::value, "");
static_assert(std::is_same<node_type1, node_type3>::value, "");
static_assert(std::is_same<node_type2, node_type3>::value, "");
```

Example. The example below uses the proposed features to write code that is fast and uses the minimum amount of memory. Many linked lists served with nodes stored in an `std::array`. It is not possible to write it portably in current C++.

```
  // My implementation of the node allocator
  using alloc_type = rt::node_allocator<int>;

  // Node type. Currently unavailable in containers.
```

```
  using node_type =
    typename std::list<int, alloc_type>::node_type;

  // Buffer for 100 elements. offset is a space reserved for
  // the allocator (three computer words in my implementation).
  std::size_t node_size = sizeof (node_type);
  std::array<char, offset + 100 * node_size> buffer = {{}};

  // Allocator instance that will serve container instances.
  alloc_type alloc(buffer);

  std::list<int, alloc_type> l1(alloc);
  std::list<int, alloc_type> l2(alloc);
  ...
  // Inserts elements. Allocation and deallocation implemented
  // with 6 lines of code (see below). Faster and simpler than
  // any allocator I could find.
  l1 = {27, 1, 60};
  ...
```

Alternative approach. For an alternative approaches, please see appendix A.

2 Motivation and scope

Consider the example from last section where a programmer needs an `std::list` with 100 integers somewhere in the code. In current C++ there are two ways to do this (if we want indeed 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. Suboptimal allocation behind `malloc`. It 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?

5. 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!)

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 like in the example from last section. This approach has a number of advantages

1. Nodes on the stack with optimal size. *You do not pay for space you do not use.*

2. Compact buffer, improving cache locality and causing minimal fragmentation.

3. No allocation overhead: I only change some pointers. (See below)

This is however not possible in the current allocator interface due to the fact that `allocate(n)` may be called with $n \neq 1$, that means I have to add unnecessary complexity to handle the many possible values of n. The size of the buffer is not anymore clear and depends on the allocation strategy/algorithm. All the added complexity is pointless. This is only one use case out of many.

General motivations. Some further motivations for this proposal are listed below.

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.

Node allocation. The simplification we are talking about comes from the fact that there is no more need for an allocation algorithm or any fancy strategy inside the allocator. We can simply build a singly linked list were the nodes have the size demanded by the container, then allocation and deallocation reduces to push and pop from the linked list.

```
pointer allocate_node()
{
  pointer q = avail; // The next free node
  if (avail)
    avail = avail->next;

  return q;
}

void deallocate_node(pointer p)
{
  p->next = avail;
  avail = p;
}
```

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].

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);
};
```

The typedef `node_allocation_only` is to ensure that container do not use array allocation on this allocator. We also propose that all node based containers expose their node types.

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 give 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 diferent 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/ccw0pTxM_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 prameter 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 approach has one advantage

1. It does not require any new member functions and typedef in the `std::allocator_traits`.

The disadvantages I see however are

1. It sounds more like a hack of the current allocator interface to achieve node allocation than a full supported feature.
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` are provided as typedef so I would prefer to keep the harmony.