Improving allocator interface for node-based containers

Document number: 0
Date: January 17, 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 through the addition of overloads of the `allocate` and `deallocate` member functions. An example implementation that explores the proposed feature 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. Other domains, like *realtime applications*, *embedded systems* or systems that aim *24/7 availability*, cannot even afford the unpredictability introduced by memory fragmentation and dynamic memory allocations in general.

To address these problems, we propose that node-based containers favor the overload `allocate()`, that can serve only one-size memory blocks, over `allocate(size_type)`, whenever `std::allocator_traits` instructs them so. The same reasoning applies to `deallocate()`. Allocations and deallocations in this case are as simple as popping and pushing from a stack, which, beyond other advantages, has constant time complexity $O(1)$. These overloads look like this

```
pointer allocate()
{
  pointer p = stack.pop();
  if (!p)
    throw std::bad_alloc();
  return p;
}

void deallocate(pointer p)
{
  stack.push(p);
}
```

which differ from their standard interfaces by the fact that they do not take the size as a parameter.

2 Motivation and scope

SOME OF the motivations behind this proposal are

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. (Unordered associative containers use sized allocations in addition to node allocation, which means they need the sized version of `allocate` as well, but for purposes other than node allocation).

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.

A question that naturally arises is: *Why not simply test whether $n = 1$ and pass allocation to an appropriate function internally?* For example

```
pointer allocate(size_type n)
{
  if (n == 1)
    return foo.allocate(); // Calls node allocation.
  return bar.allocate(n); // Calls regular allocate.
}
```

The main reason why this is an undesirable approach is that the possibility of having $n \neq 1$ means I have to handle allocations with different sizes, which is exactly what I am trying to avoid for reasons mentioned above i.e. reduce complexity, improve performance etc. Additionally, test for the condition `if (n == 1)` is pointless since it is always true on 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 node_stack::pop()
{
  pointer q = avail; // The next free node
  if (avail)
    avail = avail->next;

  return q;
}

void node_stack::push(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 that all node based containers favor the overload `allocate()` over `allocate(size_type)`, for all node allocations inside the container, whenever `std::allocator_traits` instructs them so, by means of a new typedef

```
template<class Alloc>
struct allocator_traits {
  ...
  using use_node_allocation = typename Alloc::
    use_node_allocation;
};
```

When `use_node_allocation` is not present in the allocator, the typedef should default to `std::false_type`.

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`

Pure node-based: As a result of this proposal all node based containers mentioned above, with the exception of the unordered ones, should support allocators that provide only the overload `allocate()`.

Hybrid: Unordered containers have to rebind twice, once for node allocation and once for other internal data structures. The rebound type used for node allocations should prefer the `allocate()` overload.

4 Technical Specifications

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)