Improving allocator interface for node-based containers

Document number: 0
Date: 2016-01-05
Project: Programming Language C++
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 through the addition of overloads of the `allocate` and `deallocate` member functions. An example implementation that explores the proposed feature is provided together with benchmarks.

Contents

Size management adds undue difficulties
and inefficiencies to any allocator design.
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]. 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 it is available on the allocator. The same reasoning is 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 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. The majority of containers do not manage sizes but pointlessly demand this feature from their allocators. They are: all ordered associative containers plus `std::list` and `std::forward_list`. The unordered associative containers will also benefit from the new overloads since they also do node allocation internally.

3. Most allocators found in the literature are overly complicated as a result of having to handle blocks with different sizes.

4. Support hard-realtime allocation for node-based containers. This is highly desirable to improve C++ use in embedded systems.

5. State of the art allocators like `boost::node_allocator` [3] achieve great performance gains optimizing for the $n = 1$ case.

6. 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.

7. 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.

8. This proposal makes it easy to work with pre-allocated and pre-linked nodes.

Frequently asked questions: Some of the questions the reader may be asking are:

- *Why don't you simply test whether $n = 1$ and pass allocation to your allocator? For example*

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

There are a couple of reasons why this is an undesirable approach.

1) 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.

2) Containers in major libraries always use $n = 1$, which means the condition `if (n == 1)` is an unnecessary overhead.

3) All that most node-based containers need from their allocator is space for a single node, asking for more than that means allocation logic is being built on the container itself and not on the allocator, with unpredictable side effects that users may want to avoid. If more than one node must be allocated at once for any reason, it still can loop over `allocate()`.

- *How does this proposal simplify allocator design?*

  All the simplification 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 [1, 2] for example.

For an allocator that explores the feature proposed here, please see the project [5]. For a general talk on allocators and why size management is a problem [6]. For related proposal, please see [4].

3 Impact on the Standard

THIS PROPOSAL does not require any breaking change. We require that all node based containers favor the overload `allocate()` over `allocate(size_type)` for all node allocations inside the container, whenever this member function is available. The same reasoning applies to `deallocate(pointer)` over `deallocate(pointer, size_type)`

In order to know whether the overload is available, users are requested to add a new member to the specialization of `std::allocator_traits`

```
using use_node_allocator = std::true_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] Chandler Carruth: *Efficiency with Algorithms, Performance with Data Structures* (https://www.youtube.com/watch?v=fHNmRkzxHWs)

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

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

[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2045.html

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

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