Paper Number: P1731R1
Title: Memory helper functions for Containers
Authors: Ilya Burylov <ilya.burylov@intel.com>
Ruslan Arutyunyan <ruslan.arutyunyan@intel.com>
Pablo Halpern <phalpern@halpernwightsoftware.com>

Audience: LEWG-I (Library Evolution Working Group-Incubator)
Date: 2019-10-07

I. Introduction

There are use cases when heap memory allocation should be avoided for the various reasons, which leads to loss of availability guarantees and unpredictable execution time. An example of such a use case is in safety critical applications, where dynamic memory allocation is highly restricted by respective industry standards.

Standard containers allow replacing the default heap-based `std::allocator` with a custom allocator, which shall fulfill Allocator named requirement, and that custom allocator may replace heap-based allocation with a different allocation source. Although, this mechanism allows replacing the allocator, the information required to calculate the total amount of memory to be consumed by a container and other useful information required to configure custom allocator is hidden within the specific implementation of the container. Users need to apply reverse engineering of the specific implementation to correctly configure their allocator. Furthermore, one implementation of the other standard library might require a different amount of memory than the other one for the same use-case and the problem size.

We propose an additional API that describes container allocation guarantees, in order to enable calculation of total required memory size and provide additional data required to configure specific allocation strategies.

II. Problem description and Scope

Suppose user has use-case with `std::deque` that is parametrized with some custom allocator:

```
template <typename T, typename Allocator>
void process(const T& t, const Allocator& alloc)
{
    // fill it with 1000 elements
    std::deque<T, Allocator> d(1000, t, alloc);
    // at this point it is using memory blocks.
    for (auto i = 0; i < 100000; ++i)
    {
        d.push_front(t + 3); // make a new element on the front
        d.pop_back(); // pop one off the back.
    }
}
```

User has no idea of how much memory would be required to fulfill `std::deque` needs for the example above. Thus, user does not know how to configure `Allocator` instance.

C++17 introduced a `pmr` namespace with `polymorphic_allocator` and configurable memory resources that represent different allocation strategies. But user still does not have enough information of allocation patterns of the particular STL container. Thus, user has no information how to configure the memory resource. The example below shows it:

```
int process()
{
    constexpr std::size_t buffer_size = ?; // how much memory should we
                                              pre-allocate;
    std::byte buffer[buffer_size];          // preallocated memory


    std::pmr::monotonic_buffer_resource resource{ buffer, buffer_size,
        std::pmr::null_memory_resource() }; // enough memory assertion

    std::pmr::list<int> list_var(&resource);

    list_var.push_back({});
    list_var.push_back({});
    list_var.push_back({});
}
```

In case of polymorphic allocator the standard does not answer the question how much memory is required to fulfill container memory expectations and how to guarantee that resource will not call upstream as a fallback meaning that the pre-allocated memory is enough.

So, the problem exists for both custom allocators and standardized allocator interfaces.

The current paper focuses on an API to reveal allocation patterns underneath the implementation of any given container, which would provide enough information to configure an allocator.

API changes that would be required to achieve the goal of avoiding heap memory allocation is out of scope for this proposal, but is lightly touched on in this paper.

The main goal of this paper is to get feedback on the approach and critical areas of further generalization.

III. Describing an allocation pattern

Analysis of typical containers revealed several high-level allocation patterns:

- List/Map/Set
  - Many allocations with the same block size
- Vector/String
  - Many allocations of increasing size, but no more than 2 blocks will be non-freed at any given time
- Deque/Unordered_map/Unordered_set
  - 2 very distinct allocation patterns applied in parallel
    - Many allocation of small block sizes
    - Few allocation of bigger sizes

We developed the following schema in order to be able to describe the variety of patterns:

Allocation pattern can be divided in a set of sub-patterns.

The following structure describes a unit sub-pattern of the allocations:

```
struct memory_config
```

```
{
    std::size_t max_block_size;
    std::size_t concurrent_n;
    std::size_t total_n;
    std::size_t alignment
};
```

`max_block_size` – container shall guarantee that each allocation is less than or equal to this value
`concurrent_n` – the number of simultaneously allocated blocks that are not freed
`total_n` – the number of total block allocations for the whole Container instance lifetime
`alignment` – alignment of allocated blocks

The set of sub-patterns is described as a tuple: `std::tuple<memory_config, ...>`. The estimate of the total memory requirements is a combination of requirements described by each tuple element.

IV. Describing container use cases

Most containers have no upper limit for memory allocation if usage is not restricted in some way. At the same time, the allocation pattern may depend on the way by which you limit the usage of a given container. Here is an example for `std::vector`:

If we restrict usage of `std::vector` only by limiting the length, we are unable to predict `total_n` in the descriptor.

If we additionally ban the `shrink_to_fit` call, we can estimate `total_n`, but we should assume the worst case – `std::vector` was created small and was increased slowly by resizing by one element, thus `total_n` will be ~ log2(max_length).

If we additionally require a call to `reserve()` immediately after `std::vector` default construction, we can estimate `total_n = 1` and `concurrent_n = 1` (while in other cases `concurrent_n = 2`).

That difference led us to add a notion of *container use cases* into the API.

Possible set of use-cases for the standard containers is on the table below:

Container	Use case	Restrictions	total_n	concurrent_n	max_block_size
vector string	max_size	size & reserve <= MAX_N	?	2	O(sizeof(value_type) * MAX_N)
	max_size_no_shrink	max_size plus: no shrink_to_fit	O(MAX_N)	2	O(sizeof(value_type) * MAX_N)
	max_size_no_resize	max_size _no_shrink plus: no more than 1 resize/reserve	O(log2(MAX_N))	2	O(sizeof(value_type) * MAX_N)
	max_size_reserve	max_size _no_shrink plus: one reserve() after construction	1	1	O(sizeof(value_type) * MAX_N)
[multi]map, [multi]set, forward_list, list	max_size	size <= MAX_N	?	MAX_N	O(sizeof(node_type<value_type>))
	max_element_insertions	no more element insertions	MAX_N	MAX_N	O(sizeof(node_type<value_type>))
deque	max_size	size <= MAX_N	? (Possible for some cases)	O(MAX_N) + C	max(sizeof(void*) * O(MAX_N), internal_calc(sizeof(T)))
unordered_[multi]set unordered_[multi]map	max_size*	size & reserve < max_n && max_load_factor >= passed_load_factor && rehash <= max_n / passed_load_factor	?	O(MAX_N / load_factor) + C	max(sizeof(void*) * O(MAX_N / load_factor), O(sizeof(T)))
	max_element_insertions	no more element insertions	TBD	TBD	TBD
	max_element_insertions_reserve	no more element insertions plus: one reserve() after construction	TBD	TBD	TBD

V. Proposed API approaches to query allocation pattern

The main idea is to be able to query how much memory is required for a particular container with a particular use-case and problem size using a specific memory allocation strategy. The API should be `constexpr` to allow getting the required memory size at compilation time if all of the parameters are known.

Proposed API

The proposal is to introduce the new class template in namespace `std` named `memory_helper` with two template parameters. `memory_helper` shall provide a `std::tuple` of `memory_config`'s for the specified container and use-case:

```
template <typename UseCase, typename Container>
struct memory_helper
```

`UseCase` – a tag that describes the use-case for the container
`Container` – container type

Use-case is a tag (empty class) representing the use-case for the container. All use cases would be introduced in `std::usecase` namespace.

`memory_helper` should have specialization for each container and applicable use-case for that container. For example:

```
template <typename T, typename Alloc>
struct memory_helper<std::usecase::max_element_insertions, std::list<T,
Alloc>>
```

Each specialization of `memory_helper` must provide the following members:

`config` – public data member that has `std::tuple<memory_config,…>` type.

`memory_helper(/* each specialization specific args */)` - constructor initializing the config member.

Example:

```
template <typename T, typename Alloc>
struct memory_helper<std::usecase::max_element_insertions, std::list<T,
Alloc>>
{    constexpr memory_helper(std::size_t N)
       : config(memory_config{
                 get_max_sizeof_chunk_for_list<T>(), N, N})
    {
    }

    const std::tuple<memory_config> config;
};
```

VI. Usage examples

Let us look at the solution for `std::deque` example:

```
constexpr std::size_t allocator_overhead = //allocator specific overhead;
constexpr auto max_size = 1001;
constexpr std::memory_helper<std::usecase::max_size, std::deque<T>>
                                             helper{max_size};
constexpr auto config = std::get<0>(helper.config);
constexpr auto memory_size = config.concurrent_n * config.max_block_size
                             + allocator_overhead;
const auto memory = std::make_unique<std::byte[]>(memory_size);
//allocator specific code
mem::memory_pool<config.concurrent_n, config.max_block_size,
                    config.alignment> pool(memory.get(), memory_size);
mem::custom_allocator<T, decltype(pool)> alloc(pool);

std::deque<T, decltype(alloc)> d(max_size - 1, t, alloc); // some type T

for (int i = 0; i < 100000; ++i)
{
    d.push_front(T(t)); // make a new element on the front
    d.pop_back(); // pop one off the back.
}
```

The solution works for `std::pmr::monotonic_buffer_resource` problem as well:

```
constexpr memory_helper< std::usecase::max_element_insertions,
std::list<int>> h{3};

constexpr memory_config mc{std::get<0>(h.config)};

constexpr std::size_t buffer_size = mc.max_block_size *
    mc.total_n;
std::byte buffer[buffer_size]{};
auto& null_resource = std::pmr::null_memory_resource();
std::pmr::monotonic_buffer_resource resource{buffer, buffer_size,
    null_resource};
std::pmr::list<int> list_var(&resource);

list_var.push_back({});
list_var.push_back({});
list_var.push_back({});

list_var.push_back({}); // Out of memory
```

VII. Further investigation in plans

a) Nested containers API

The case of `std::list<std::string>` implies allocation on several levels, which would require additional information to configure the `memory_resource`. Early internal investigation showed potential for generalization of the API to recurrent level. Details are targeted to the next iteration of the paper.

b) Allocators contact API

Need to define an API that could tell the user the allocation strategy overhead if any (e.g. strategy of pool replenishment from the upstream), but that should be a separate paper.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804