Paper Number: P1731R0
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 (Library Evolution Working Group)
Date: 2019-06-03

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 allocation strategy, the information required to calculate the total amount of memory to be consumed by a container and other details 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 another 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. Motivation and Scope

C++17 introduced a `pmr` namespace with `polymorphic_allocator` and different memory resources that represent several allocation strategies.

That became a step forward to address a problem of providing a standard mechanism to configure underlying allocation mechanism for a given use case, but it does not address the other part of the story: data required to configure some memory resource.

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 a memory resource.

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 initial feedback on the feasibility of the approach and get an advice on critical areas of further approach generalization. A proposal along these lines would be targeted for C++23.

III. Problem description

Let us consider the following code snippet:

```
constexpr std::size_t buffer_size = ? // how much memory should we pre-allocate;
```

```cpp
std::byte buffer[buffer_size]{}; // preallocated memory
auto null_resource = std::pmr::null_memory_resource(); // assertion that memory is enough

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({});
```

The main question is how much memory would be required to fulfill container memory expectations and guarantee that resource won't call upstream as a fallback meaning that the pre-allocated memory is enough?

IV. 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:

```cpp
struct memory_config
{
    std::size_t max_block_size;
    std::size_t concurrent_n;
    std::size_t total_n;
};
```

`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

The set of sub-patterns is described as a tuple: `std::tuple<memory_config, ...>`. The tuple means that allocations describing all elements of the pattern can be interleaved. That means that the estimate of the total memory requirements is a sum of requirements described by each element.

V. 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.

Most use-cases are container-specific: e.g. shrink_to_fit is specific to vector/deque/string and irrelevant to everything else. Limitation on max_load_factor is relevant only to unordered containers, where it is critical to calculate memory consumption.

VI. Possible approaches to query allocation pattern data

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.

a) Standalone 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::monotonic_growth, 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::monotonic_growth, 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;
};
```

b) Use case as a part of the container

We don't introduce `memory_helper` API and don't do use-case tags. Instead, we make memory helper and use-case the same instance and make additional aliases for implementation-defined classes inside each Container for every applicable use-case. E.g.

```
template<typename T, typename Alloc> class list

{

    ...

    using value_type;

    ...

    using monotonic_growth_helper = /* implementation defined */;

}
```

`monotonic_growth_helper` (as all other helpers) shall provide the following API:

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

`monotonic_growth_helper (/* each helper specific args */)` - constructor initialized config member with dedicated `memory_config`.

Example:

```
constexpr std::list<int>::monotonic_growth_helper h{6};
```

However, this approach gives impossibility to write partial template specialization with specific helper Consider the following example:

```
template <typename T> struct A{};
```

We cannot write something like that:

```
template <typename T, typename Alloc>

struct A<typename std::list<T,Alloc>::monotonic_growth_helper>

{};
```

VII. Usage examples

Let's consider the possible solution with `std::pmr::monotonic_buffer_resource.`

a) Standalone API

```
constexpr memory_helper< std::usecase::monotonic_growth, 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.concurrent_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
```

b) Part of the container API

```
constexpr std::list<int>>::monotonic_growth_helper h{3};

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

constexpr std::size_t buffer_size = mc.max_block_size *
    mc.concurrent_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
```

VIII. 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) Different use-case description

We are going to propose an initial set of use cases for existing standard containers.

c) 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)

d) Potential solution for variable-length local arrays

The C++ standard committee has struggled to define a way to support variable-length arrays, as the C approach is insufficient to support user-defined types (and is counter to C++ style in other ways). Conceivably, an approach to VLAs could be based on the facilities proposed in this paper.

IX. Additional extensions required

This API serves well for both monotonic and pool allocation strategies. Unfortunately, current `pool_resource` implementation has lack of configurability and implementation specification required to provide a guarantee on memory consumption.

Legal Disclaimer & Optimization Notice

Optimization Notice