The Concept of Memory Allocator

Document number:	P1172R1
Date:	2019-06-16
Project:	Programming Language C++
Audience:	LEWG, LWG

Authors:	Mingxin Wang (Microsoft (China) Co., Ltd.)
Reply-to:	Mingxin Wang <mingxwa@microsoft.com>

Table of Contents

1 History

1.1 Changes from P1172R0

- change the required expression for the concepts from member function to member function template, and
- change the name of the class `**`memory_allocator`**` into `**`global_memory_allocator`**`, and
- add `const` qualifier to the member function templates of `**`global_memory_allocator`**`.

2 Introduction

Runtime memory allocation is a common requirement in C++. This paper proposed a new way to abstract this concept in order to simplify the code requiring runtime memory allocation, especially in type-erased contexts.

The **`BasicMemoryAllocator`** requirements and the class **`global_memory_allocator`** proposed in this paper are used in the implementation for the Concurrent Invocation library [P0642R2] and the PFA [P0957R2].

3 Motivation

In C++98, the concept of "Allocator" was introduced, and was widely adopted in STL. However, since the concept of "Allocator" has too many customization points and is type-specific, it becomes difficult to reuse this concept in type-erased components. For example, the class template **`std::function`** used to have a constructor that allow customized allocator types, just like other STL containers does, but the constructor was eventually removed in C++17 because "the semantics are unclear, and there are technical issues with storing an allocator in a type-erased context and then recovering that allocator later for any allocations needed during copy assignment" [P0302R1]. With the proposed concept of "Memory Allocator", it will be much easier to add customization points in type-erased components, like the "PFA" - a generic, extendable and efficient solution for polymorphic programming [P0957R2].

In C++17, we have the concept of "Memory Resources". However, it is defined with "intrusive polymorphism", and any implementation of the concept shall be derived from the base class **`std::pmr::memory_resource`** with virtual functions, which will introduce unnecessary runtime space occupation and overhead when the "memory resource" itself is not required to be polymorphic. Besides, since the size and alignment of the type to allocate are known at compile-time, it will be better for performance to pass them to the corresponding functions as compile-time constants rather than runtime variables. Considering usability and performance, not only does the "Memory Allocator" not require the implementations to be polymorphic themselves, but also allow passing size and alignment with compile-time constants.

Therefore, if the concept of "Memory Allocator" is introduced in the standard, it will become possible for us to introduce reliable and extendable memory allocation mechanism to type-erased components, including the class template **`std::function`**, the class **`std::any`**.

4 Technical Specification

4.1 Requirements for Memory Allocator types

4.1.1 `BasicMemoryAllocator` requirements

A type **`MA`** meets the **`BasicMemoryAllocator`** requirements of specific values **`SIZE`** and **`ALIGN`** of type `` `constexpr std::size_t` `` if the following expressions are well-formed and have the specified semantics (`` `ma` `` denotes a value of type `` `MA` ``).

`ma.template allocate<SIZE, ALIGN>()`

Effects: Allocates a continuous block of memory with at least specific size of **`SIZE`** and alignment of **`ALIGN`**. The allocated memory will be available until a subsequent corresponding call to **`ma.template deallocate<SIZE, ALIGN>(p)`** where `` `p` `` is the return value from this function.

Return type: convertible to **`void*`**

Returns: A pointer pointing to the first byte of the memory being allocated.

`ma.template deallocate<SIZE, ALIGN>(p)`

Requires: **`p`** shall have been returned from a prior call to **`ma.template allocate<SIZE, ALIGN>()`**, and the

storage at **p** shall not yet have been deallocated.

Effects: Deallocates the memory pointed by **p** allocated before.

4.1.2 `MemoryAllocator` requirements

A type **MA** meets the **MemoryAllocator** requirements of specific values **SIZE** and **ALIGN** of type `**constexpr std::size_t**` if it meets the **BasicMemoryAllocator** requirements of **SIZE** and **ALIGN**, and the following expressions are well-formed and have the specified semantics (`**ma**` denotes a value of type `**MA**`).

ma.template allocate<SIZE, ALIGN>(n)

Effects: Allocates a continuous block of memory with at least specific size of **(n * SIZE)** and alignment of **ALIGN**. The allocated memory will be available until a subsequent corresponding call **ma.template deallocate<SIZE, ALIGN>(p, n)** where `p` is the return value from this function.

Return type: **void***

Returns: A pointer pointing to the first byte of the memory being allocated.

ma.template deallocate<SIZE, ALIGN>(p, n)

Requires: **p** shall have been returned from a prior call to **ma.template allocate<SIZE, ALIGN>(n)**, and the storage at **p** shall not yet have been deallocated.

Effects: Deallocates the memory pointed by **p** allocated before.

4.2 Class `global_memory_allocator`

```
namespace std {

class global_memory_allocator {
 public:
  template <size_t SIZE, size_t ALIGN>
  void* allocate() const;

  template <size_t SIZE, size_t ALIGN>
  void deallocate(void* p) const;

  template <size_t SIZE, size_t ALIGN>
  void* allocate(size_t n) const;

  template <size_t SIZE, size_t ALIGN>
  void deallocate(void* p, size_t n) const;
};

}
```

The class **`global_memory_allocator`** meets the `**MemoryAllocator**` requirements for any valid `**SIZE**` and `**ALIGN**` of type `**constexpr std::size_t**`, and is expected to be included in header `**<memory>**`.