

The Concept of Memory Allocator

Document number: P1172R0
Date: 2018-10-03
Project: Programming Language C++
Audience: LEWG, LWG
Authors: Mingxin Wang
Reply-to: Mingxin Wang <wmx16835vv@163.com>

Table of Contents

The Concept of Memory Allocator.....	1
1 Introduction.....	1
2 Technical Specification.....	2
2.1 Requirements for Memory Allocator types.....	2
2.1.1 BasicMemoryAllocator requirements.....	2
2.1.2 MemoryAllocator requirements	2
2.2 Class memory_allocator	3
3 Known Use Case.....	3

1 Introduction

Runtime memory allocation is a common requirement in C++ projects. 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.

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 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 [\[P0957R1\]](#).

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` and the "PFA" [\[P0957R1\]](#).

2 Technical Specification

2.1 Requirements for Memory Allocator types

2.1.1 BasicMemoryAllocator requirements

A type **MA** meets the **BasicMemoryAllocator** requirements of specific `std::integral_constant` of **SIZE** and **ALIGN** if the following expressions are well-formed and have the specified semantics (**ma** denotes a value of type **MA**, **s** donates a value of `std::integral_constant<std::size_t, SIZE>`, **a** donates a value of `std::integral_constant<std::size_t, ALIGN>`, **p** donates the returned value).

ma.allocate(s, a)

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.deallocate(p, s, a)**.

Return type: `void*`

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

ma.deallocate(p, s, a)

Requires: **p** shall have been returned from a prior call to **ma.allocate(s, a)**, and the storage at **p** shall not yet have been deallocated.

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

2.1.2 MemoryAllocator requirements

A type **MA** meets the **MemoryAllocator** requirements of specific `std::integral_constant` of **SIZE** and **ALIGN** if it meets the **BasicMemoryAllocator** requirements of `std::integral_constant` of **SIZE** and **ALIGN**, and the following expressions are well-formed and have the specified semantics (**ma** denotes a value of type **MA**, **s** donates a value of `std::integral_constant<std::size_t, SIZE>`, **a** donates a value of `std::integral_constant<std::size_t, ALIGN>`, **p** donates the returned value, **n** donates the length of the array of `std::size_t`).

ma.allocate(n, s, a)

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 to **ma.deallocate(p, n, s, a)**.

Return type: `void*`

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

ma.deallocate(p, n, s, a)

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

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

2.2 Class `memory_allocator`

```
namespace std {  
  
class memory_allocator {  
public:  
    template <size_t SIZE, size_t ALIGN>  
    void* allocate(integral_constant<size_t, SIZE>,  
        integral_constant<size_t, ALIGN>);  
  
    template <size_t SIZE, size_t ALIGN>  
    void deallocate(void* p, integral_constant<size_t, SIZE>,  
        integral_constant<size_t, ALIGN>);  
  
    template <size_t SIZE, size_t ALIGN>  
    void* allocate(size_t n, integral_constant<size_t, SIZE>,  
        integral_constant<size_t, ALIGN>);  
  
    template <size_t SIZE, size_t ALIGN>  
    void deallocate(void* p, size_t, integral_constant<size_t, SIZE>,  
        integral_constant<size_t, ALIGN>);  
};  
  
}
```

The class `memory_allocator` meets the `MemoryAllocator` requirements for any positive `std::integral_constant` of `SIZE` and `ALIGN`, and is expected to be included in header `<memory>`.

3 Known Use Case

The `BasicMemoryAllocator` requirements and the class `memory_allocator` is used as an [experimental extension](#) for the PFA [P0957R1], and is used in the constructor of the built-in addresser of value semantics.