Local allocator

Montag, 22. November 2021 11:29

Links

# Andrei Alexandrescu
CppCon 2015: Andrei Alexandrescu “std::allocator...” - YouTube

# Pablo Halpern & Alisdair Meredith
https://www.youtube.com/watch?v=RLezJuqNcEQ

# John Lakos
Local (Arena) Memory Allocators Part 1 - John Lakos - Meeting C++ 2017 - YouTube
(158) Local (Arena) Allocators Part II - John Lakos - Meeting C++ 2017 - YouTube
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2035r0.pdf

# Json Turner
https://www.youtube.com/watch?v=q6A7cKFXjY0
https://www.youtube.com/watch?v=6BLlIj2QoT8
https://www.youtube.com/watch?v=vXJ1dwJ9QkI
https://www.youtube.com/watch?v=Zt0q3OEeuB0

# Richard Kaiser
https://www.youtube.com/watch?v=Ju41sGwE88w

# Benchmarking
Quick C++ Benchmarks (quick-bench.com)

Intro

The main limitation with std containers, is that Allocator is part of the container definition

```
namespace std
{
    template <class T, class Allocator = std::allocator<T>>
    class vector
    {
    };
}
```

which means that different allocator types (*Allocator*) result in different container types, even when they contains the same underlying type (*T*)

```
template <typename T>
void f(const std::vector<T>& );


std::vector<int> a {1, 2, 3};
std::vector<int, MyAllocator<int>> b(myAllocator);
…

f(a); // OK
f(b); // Error - incompatible type
```

The function template **f** expects any std::vector<T, **std::allocator<T>>** type

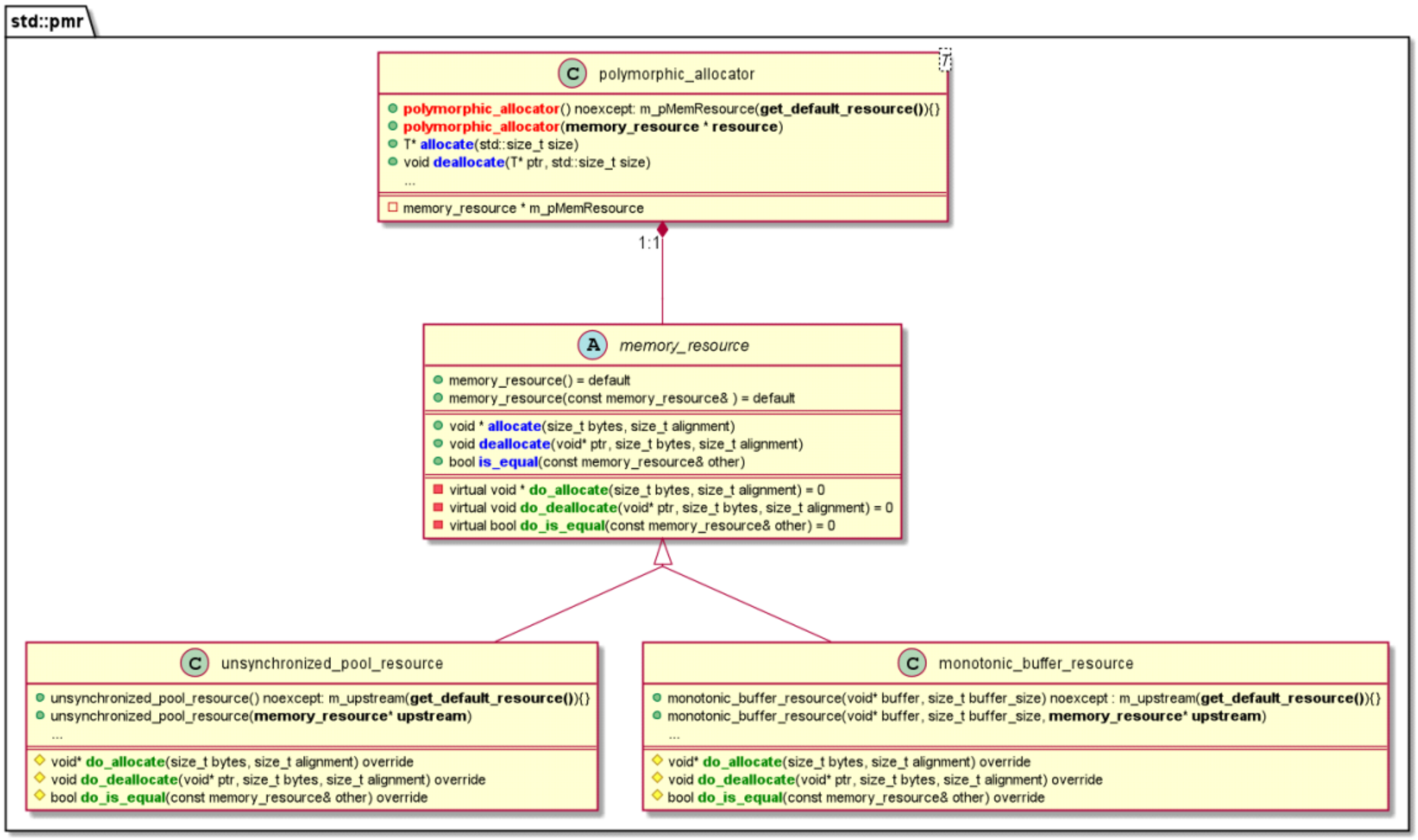
Polymorphic allocator

Motivation

- the main motivation for using custom allocators is **performance**
  - ability to (re)use contiguous memory blocks, even on the stack (*monotonic_buffer_resource*).
    This kind of "locality" that results in copy the very same block of the main memory into the cache lines (64 bytes each) is maybe even more important than the reduced number of heap allocation/deallocation and all consequences of that (like "false sharing" in multithread environment: instead one can use the std::pmr memory with thread local storage duration- thread_local)
- reduce **fragmentation** and **diffusion** constantly using new/delete
- avoid concurrency lock within the same thread
- benchmarking and statistic (logging allocation/deallocation)

Header
#include <memory_resource>

```
namespace std::pmr
{
    template <typename T>
    using vector<T> = std::vector<T, polymorphic_allocator<T>>;
}
```


std::pmr
polymorphic_allocator
polymorphic_allocator() noexcept: m_pMemResource(get_default_resource()){}
polymorphic_allocator(memory_resource * resource)
T* allocate(std::size_t size)
void deallocate(T* ptr, std::size_t size)
...
memory_resource * m_pMemResource
1:1
memory_resource
memory_resource() = default
memory_resource(const memory_resource&) = default
void * allocate(size_t bytes, size_t alignment)
void deallocate(void* ptr, size_t bytes, size_t alignment)
bool is_equal(const memory_resource& other)
virtual void * do_allocate(size_t bytes, size_t alignment) = 0
virtual void do_deallocate(void* ptr, size_t bytes, size_t alignment) = 0
virtual bool do_is_equal(const memory_resource& other) = 0
unsynchronized_pool_resource
unsynchronized_pool_resource() noexcept: m_upstream(get_default_resource()){}
unsynchronized_pool_resource(memory_resource* upstream)
...
void* do_allocate(size_t bytes, size_t alignment) override
void do_deallocate(void* ptr, size_t bytes, size_t alignment) override
bool do_is_equal(const memory_resource& other) override
monotonic_buffer_resource
monotonic_buffer_resource(void* buffer, size_t buffer_size) noexcept : m_upstream(get_default_resource()){}
monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream)
...
void* do_allocate(size_t bytes, size_t alignment) override
void do_deallocate(void* ptr, size_t bytes, size_t alignment) override
bool do_is_equal(const memory_resource& other) override

std::pmr::polymorphic_allocator type will be constructed from the **std::pmr::memory_resource** (or default allocator otherwise) implicitly, which is itself an abstract interface

```
namespace std::pmr
{
    class memory_resource
    {
      public:
        virtual ~memory_resource() = default;

        void* allocate(std::size_t bytes, std::size_t alignment);
        void  deallocate(void* ptr, std::size_t size, std::size_t alignment);
        bool  is_equal(const memory_resource& other);
      private:
        virtual void* do_allocate(std::size_t bytes, std::size_t alignment) = 0;
        virtual void do_deallocate(void* ptr, std::size_t size, std::size_t alignment) = 0;
        virtual bool do_is_equal(const memory_resource& other) = 0;
        ...
    };
}
```

Allocator implements allocation/deallocation by calling the internally stored reference to the std::pmr::memory_resource - cppreference.com
Memory_resource will (virtual) dispatch these calls to the derived class specific implementation of the interface.
There are some library supported - predefined memory resources:

- **std::pmr::unsychronized_pool_resource**
  - pools of size-tailored chunks (4 bytes, 16 bytes, 32 bytes, …)
  - good for dynamic objects (containers) of a different sizes
  - single-thread solution (the same allocator can't be used for containers in different thread context)

  @note: There is a thread-safe (locking) version: ***std::pmr::synchronized_pool_resource***

- **std::pmr::monotonic_buffer_resource**
  - For preallocation of the big chunks of memory on the **stack**
  - There is no deallocation - it can only monotonically grow (by using the upstream allocator: default heap)
    The entire contiguous block of memory will be released once, when memory resource goes out of scope
  - ultra-fast
  - single-threaded

Composing the allocator

Each memory resource can be constructed with **upstream memory resource** - the one which
will be addressed when there is no enough memory for growing (allocating).
This way, we can combine the best of them

```
std::byte buff[1024 * 1024];
std::pmr::monotonic_buffer_resource upstream(buff, sizeof buff, std::pmr::null_memory_resource());
std::pmr::unsynchronized_pool_resource alloc(&upstream);

std::pmr::vector<uint8_t> data(&alloc);
```

In case that we want to prohibit the uncontrolled memory expansion - in case of insufficient preallocated contiguous block of memory, we may use other allocation strategy, specifying for the *upstream* memory resource ***std::pmr::null_memory_resource***.
This way, if the all preallocated memory is exhausted, rather than reach for the heap - the exception will be thrown (*std::bad_alloc*)

Example #1
https://godbolt.org/z/h4xvG8vc4
https://www.quick-bench.com/q/Dvw_h-CUc3T9hy8bzFSs06SShXY
https://www.quick-bench.com/q/dryzn9PydTiEJPoaQcM2sMhVf-E

Nested containers (scoped allocator model)

In traditional way, the vector and the string will use a different allocator - as matter of fact, each string stored into vector, will use its own allocator (memory diffusion)

```
std::vector<std::string> s1;
```

This way, using the preallocated memory, the same memory block will be reused for allocation of both: vector, and nested container

```
std::array<std::byte, 1024> buff;
std::pmr::monotonic_buffer_resource rsrc(buff.data(), buff.size());

std::pmr::vector<std::pmr::string> s2(&rsrc); // implicit c-tor of polymorphic_allocator
```

#Examples 2
https://godbolt.org/z/nqP8ao6hj
https://godbolt.org/z/4Mz9Pfrez

https://godbolt.org/z/ao9efnYcY

Singleton

One can use the on the stack created allocator along with placement new to create a singleton: function -scope static instance of the object (thread-safe starting with C++11), that will be safely released upon process termination
https://godbolt.org/z/Ps55jeP61

```
template <typename T, typename... Args>
T* getSingletonPtr(Args&&... args)
{
    static std::array<std::byte, sizeof(T)> buff;
    static std::pmr::monotonic_buffer_resource mem(buff.data(), buff.size()); // static memory on the stack
    static T* singleton = new (&mem) T(std::forward<Args>(args)...);  // placement new
    return singleton;
}

template <typename T>
void releaseSingletonPtr(T*& singleton)
{
    if (singleton)
    {
        singleton->~T();
        singleton = nullptr;
    }
}
```

Custom AA (Allocator Aware) types

In case that the user-defined type
- has AA aware base class, or
- consist of the AA non-static data member types

to use the appropriate allocation policy (std::pmr::polymorphic_allocator), the fallowing needs to be done
- the alias template ***allocator_type*** needs to be linked to the polymorphic allocation policy
- c-tor overload resolution needs to be enhanced with allocator as additional argument.
- copy/move c-tors need to be enhanced with allocator as additional argument

```
template <typename T>
class A
{
    public:
        using value_type = T;
        …
        using allocator_type = std::pmr::polymorphic_allocator<std::byte>;
        A(std::initializer_list<T> data, int id, const allocator_type& alloc = allocator_type()) :
            m_data(data, alloc),
            m_id(id)
            {}
        A(std::size_t size, const T& val=T(), int id, const allocator_type& alloc = allocator_type()):
            m_data(size, val, alloc),
            m_id(id)
            {}
        ...

        // Copy-constructor
        A(const A& other, const allocator_type& alloc = allocator_type()):
            m_data(other.m_data, alloc),
            m_id(other.m_id)
            {}
        // Move-constructor
        A(A&& other, const allocator_type& alloc = allocator_type()) noexcept:
            m_data(std::move(other.m_data)),
            m_id(other.m_id)
            {}
    private:
        std::pmr::vector<T> m_data;
        int m_id;
};
```

This way we can utilized on the polymorphic allocation policy - to reuse the same preallocated memory not only for std::pmr polymorphic containers, but also for the user-defined types (std::pmr::deque<A>) consist of non-static data member with polymorphic allocators.

# Test Allocator
https://github.com/bloomberg/p1160

# Bloomberg implemntation

https://github.com/bloomberg/bde/blob/master/groups/bdl/bdlma

https://github.com/bloomberg/bde/blob/master/groups/bdl/bdlma/bdlma_buffermanager.h