Polymorphic Allocator

C++ 17 has introduced Polymorphic allocator as an enhancement to the standard allocator from the Standard Library.

WE'LL COVER THE FOLLOWING

Polymorphic Allocator

To be concise, a **polymorphic allocator** conforms to the rules of an allocator from the Standard Library.

However, at its core, it uses a memory resource object to perform memory management.

> Polymorphic Allocator contains a pointer to a memory resource class, and that's why it can use a virtual method dispatch. You can change the memory resource at runtime while keeping the type of the allocator.

All the types for polymorphic allocators live in a *separate namespace* `std::pmr` (PMR stands for Polymorphic Memory Resource), in the `<memory_resource>` header.

Core elements of `pmr`:

- `std::pmr::memory_resource` - is an abstract base class for all other implementations. It defines the following pure virtual methods: `do_allocate`, `do_deallocate` and `do_is_equal`.

- `std::pmr::polymorphic_allocator` - is an implementation of a standard allocator that uses `memory_resource` object to perform memory allocations and deallocations.
- global memory resources accessed by `new_delete_resource()` and `null_memory_resource()`
- a set of predefined memory pool resource classes:
  - `synchronized_pool_resource`
  - `unsynchronized_pool_resource`
  - `monotonic_buffer_resource`
- template specialisations of the standard containers with polymorphic allocator, for example `std::pmr::vector`, `std::pmr::string`, `std::pmr::map` and others. Each specialisation is defined in the same header file as the corresponding container.

Predefined memory resources

Here's a short overview of the predefined memory resources:

Resource	Description
`new_delete_resource()`	a free function that returns a pointer to a global "default" memory resource. It manages memory with the global `new` and `delete`
`null_memory_resource()`	a free function that returns a pointer to a global "null" memory resource which throws `std::bad_alloc` on every allocation
`synchronized_pool_resource`	thread-safe allocator that manages pools of different sizes. Each pool is a set of chunks that are divided into blocks of uniform size.

`unsynchronized_pool_resource`	non-thread-safe `pool_resource`
`monotonic_buffer_resource`	non-thread-safe, fast, special-purpose resource that gets memory from a preallocated buffer, but doesn't release it with deallocation.

> It's also worth mentioning that pool resources (including `monotonic_buffer_resource`) can be chained. So that if there's no available memory in a pool, the allocator will allocate from the "upstream" resource.

Let's look at a simple example of `monotonic_buffer_resource` and `pmr::vector`:

```
#include <iostream>
#include <memory_resource>
#include <vector>

int main() {
    char buffer[64] = {};
    std::fill_n(std::begin(buffer), std::size(buffer)-1, '_');
    std::cout << buffer << '\n';

    std::pmr::monotonic_buffer_resource pool{std::data(buffer), std::size(buffer)};

    std::pmr::vector<char> vec{&pool};
    for (char ch='a'; ch <= 'z'; ++ch)
        vec.push_back(ch);

    std::cout << buffer << '\n';
}
```

In the above example, we use a monotonic buffer resource, initialized with a memory chunk from the stack. By using a simple `char buffer[]` array, we can easily print the contents of the "memory". The vector gets memory from the pool, and if there's no more space available, it will ask for memory from the "upstream" resource (which is `new_delete_resource` by default). The example shows vector reallocations when there's a need to insert more elements. Each time the vector gets more space, so it eventually fits all of the letters.

More Information

This section briefly discussed the idea of polymorphic allocators and memory resources. If you want to learn more about this topic, there's a long chapter in C++17 The Complete Guide by Nicolai Josuttis.

There are also several conference talks, for example Allocators: The Good Parts by Pablo Halpern from CppCon 2017.

> **Extra Info:** See more information in P0220R1 and P0337R0

Let's look at the compiler support for the topics we have covered in this section.