C++17's std::pmr Comes With a Cost

C++Now 2018
David Sankel
@david_sankel
Bloomberg

`pmr` sandbox

https://goo.gl/Pe5Zy1

https://github.com/camio/pmr_sandbox.git

Two themes

- How to use pmr
- Conversation on its implications

Problems `std::pmr` attempts to solve

Allocation is slow

Fragmentation hurts performance

Pre-C++17 allocators are overly complicated

Speed

- `synchronized_pool_resource`
  - For data frequently accessed at the same time
- `unsynchronized_pool_resource`
  - no thread synchronization cost
- `monotonic_buffer_resource`
  - high speed, high space

Simplicity

```
#include <memory_resource>

class LoggingResource : public std::pmr::memory_resource {
 public:
  LoggingResource(std::pmr::memory_resource *underlyingResource)
    : d_underlyingResource(underlyingResource)
  {
  }
 private:
  std::pmr::memory_resource *d_underlyingResource;

  void *do_allocate(size_t bytes, size_t align) override
  {
    std::cout << "Allocating " << bytes << " bytes" << std::endl;
    return d_underlyingResource->allocate(bytes, align);
  }
  void do_deallocate(void *p, size_t bytes, size_t align) override
  {
    return d_underlyingResource->deallocate(p, bytes, align);
  }
  bool do_is_equal(memory_resource const& other) const noexcept override
  {
    return d_underlyingResource->is_equal(other);
  }
};
```

```cpp
int main()
{
    static LoggingResource memoryResource{std::pmr::new_delete_resource()};
    std::pmr::set_default_resource(&memoryResource);

    std::cout << "## vector<int> test" << std::endl;
    std::pmr::vector<int> ints;
    ints.push_back(33);
    ints.push_back(34);
}
```

↓

```
## vector<int> test
Allocating 4 bytes
Allocating 8 bytes
```

```cpp
class Bar {
    std::string data{"data"};
};

class Foo {
    std::unique_ptr<Bar> d_bar{std::make_unique<Bar>()};
};

// ...

std::cout << "\n## vector<Foo> test" << std::endl;
std::pmr::vector<Foo> foos;
foos.emplace_back();
foos.emplace_back();
```

↓

```
## vector<Foo> test
Allocating 8 bytes
Allocating 16 bytes
```

```
class Bar2 {
    std::string data{"data"};
};

class Foo2 {
    std::unique_ptr<Bar2, polymorphic_allocator_delete> d_bar;

  public:
    Foo2()
      : d_bar(nullptr, {{std::pmr::get_default_resource()}})
    {
        std::pmr::polymorphic_allocator<Bar2> alloc{
            std::pmr::get_default_resource()};
        Bar2 *const bar = alloc.allocate(1);
        alloc.construct(bar);
        d_bar.reset(bar);
    }
};
```

```cpp
class Bar2 {
    std::string data{"data"};
};

class Foo2 {
    std::unique_ptr<Bar2, polymorphic_allocator_delete> d_bar;

  public:
    Foo2()
      : d_bar(nullptr, {{std::pmr::get_default_resource()}})
    {
        std::pmr::polymorphic_allocator<Bar2> alloc{
            std::pmr::get_default_resource()};
        Bar2 *const bar = alloc.allocate(1);
        try {
            alloc.construct(bar);
        } catch(...) {
            alloc.deallocate(bar, 1);
            throw;
        }
        d_bar.reset(bar);
    }
};
```

```cpp
class polymorphic_allocator_delete {
  public:
    polymorphic_allocator_delete(
                          std::pmr::polymorphic_allocator<std::byte> allocator)
      : d_allocator(std::move(allocator))
    {
    }
    template <typename T>
    void operator()(T *tPtr)
    {
        std::pmr::polymorphic_allocator<T>(d_allocator).destroy(tPtr);
        std::pmr::polymorphic_allocator<T>(d_allocator).deallocate(tPtr, 1);
    }

  private:
    std::pmr::polymorphic_allocator<std::byte> d_allocator;
};
```

```cpp
class Bar2 {
    std::string data{"data"};
};

class Foo2 {
    std::unique_ptr<Bar2, polymorphic_allocator_delete> d_bar;

  public:
    Foo2()
      : d_bar(nullptr, {{std::pmr::get_default_resource()}})
    {
        //...
    }
};

//...

std::cout << "\n## vector<Foo2> test" << std::endl;
std::pmr::vector<Foo2> foo2s;
foo2s.emplace_back();
foo2s.emplace_back();
```

↓

```
## vector<Foo2> test
Allocating 16 bytes <- was 8
Allocating 24 bytes
Allocating 32 bytes <- was 16
Allocating 24 bytes
```

```
class Bar3 {
    std::pmr::string data{"data"};
//       ^^^
};

class Foo3 {
    std::unique_ptr<Bar3, polymorphic_allocator_delete> d_bar;

  public:
    Foo3(); // same as before
};

//...

std::cout << "\n## vector<Foo3> test" << std::endl;
std::pmr::vector<Foo3> foo3s;
foo3s.emplace_back();
foo3s.emplace_back();
```

↓

```
## vector<Foo3> test
Allocating 16 bytes
Allocating 32 bytes <- was 24
Allocating 32 bytes
Allocating 32 bytes <- was 24
```

```
class default_polymorphic_allocator_delete {
  public:
    template <typename T>
    void operator()(T *tPtr)
    {
        std::pmr::polymorphic_allocator<T>(std::pmr::get_default_resource())
            .destroy(tPtr);
        std::pmr::polymorphic_allocator<T>(std::pmr::get_default_resource())
            .deallocate(tPtr, 1);
    }
};
```

```
struct Bar4 {
  public:
    std::pmr::string data{"data"};
};
class Foo4 {
  public:
    std::unique_ptr<Bar4, default_polymorphic_allocator_delete> d_bar;

    Foo4()
    {
        std::pmr::polymorphic_allocator<Bar4> alloc{
            std::pmr::get_default_resource()};
        Bar4 *const bar = alloc.allocate(1);
        alloc.construct(bar);
        d_bar.reset(bar);
    }
};
```

↓

```
## vector<Foo4> test
Allocating 8 bytes  <- back to 8 from 16
Allocating 32 bytes
Allocating 16 bytes <- back to 16 from 32
Allocating 32 bytes
```

```cpp
void f() {
  static Foo4 foo4;
}

void main() {
  f();
  std::pmr::set_default_resource( /* ... */ );
}
```

Allocator awareness

```
class Bar6 {
  public:
    Bar6(std::pmr::polymorphic_allocator<std::byte> allocator = {})
    : data("data", allocator)
    {
    }

  private:
    std::pmr::string data;
};
```

```
class Foo6 {
    std::pmr::polymorphic_allocator<std::byte>          d_allocator;
    //^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    // New. We're now storing the allocator locally so we can use it later.

    std::unique_ptr<Bar6, polymorphic_allocator_delete> d_bar;

    //...
};
```

```
class Foo6 {
    //...
  public:
    typedef std::pmr::polymorphic_allocator<std::byte> allocator_type;
    //^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    //New. Required for 'std::vector' to realize this is allocator aware.

    std::pmr::polymorphic_allocator<std::byte> get_allocator()
        // Return the allocator this object was constructed with.
    {
        return d_allocator;
    }
    //...
};
```

```
class Foo6 {
    //...
  public:
    //...
    Foo6(std::pmr::polymorphic_allocator<std::byte> allocator = {})
    : d_allocator(allocator)
    , d_bar(nullptr, {allocator})
    {
        std::pmr::polymorphic_allocator<Bar6> barAlloc{allocator};
        Bar6 *const                           bar = barAlloc.allocate(1);
        try {
            barAlloc.construct(bar, allocator);
            //                      ^^^^^^^^^
            //                      New
        } catch(...) {
            alloc.deallocate(bar, 1);
            throw;
        }
        d_bar.reset(bar);
    }
    //...
};
```

```
class Foo6 {
    //...
  public:
    //...
    Foo6(const Foo6&                                other,
         std::pmr::polymorphic_allocator<std::byte> allocator = {})
    : Foo6(allocator)
    {
        *d_bar = *other.d_bar;
    }

    Foo6& operator=(const Foo6& other)
    {
        *d_bar = *other.d_bar;
        return *this;
    }
    //...
};
```

```
class Foo6 {
    //...
  public:
    //...
    Foo6(Foo6&& other,
         std::pmr::polymorphic_allocator<std::byte> allocator = {})
    : d_allocator(allocator)
    , d_bar(nullptr, {d_allocator})
    {
        if(get_allocator() == other.get_allocator())
            d_bar.reset(other.d_bar.release());
        else {
            std::pmr::polymorphic_allocator<Bar6> barAlloc{allocator};
            Bar6 *const bar = barAlloc.allocate(1);
            try {
              barAlloc.construct(bar, allocator);
            } catch (...) {
              barAlloc.deallocate(bar, 1);
              throw;
            }
            d_bar.reset(bar);
            operator=(other);
        }
    };
};
```

```
class Foo6 {
  public:
    //...
    Foo6(std::pmr::polymorphic_allocator<std::byte> allocator =
             std::pmr::get_default_resource());

    Foo6(const Foo6&                                other,
         std::pmr::polymorphic_allocator<std::byte> allocator = {});

    Foo6& operator=(const Foo6& other);

    Foo6(Foo6&& other,
         std::pmr::polymorphic_allocator<std::byte> allocator = {});
};
//...
std::cout << "\n## vector<Foo6> test" << std::endl;
std::pmr::vector<Foo6> foo6s(
                   std::pmr::polymorphic_allocator<Foo6>{&memoryResource});
foo6s.emplace_back();
foo6s.emplace_back();
```

↓

```
## vector<Foo6> test
Allocating 24 bytes <- originally 8
Allocating 32 bytes <- originally 24
Allocating 48 bytes
Allocating 32 bytes
Allocating 32 bytes <- What is this?
```

Strong Exception Safety

`pmr`'s dirty little secret

```
class Foo7 {
  public:
    //...
    Foo7(Foo7&& other,
         std::pmr::polymorphic_allocator<std::byte> allocator);
//                     New. Removed default argument        ^

    Foo7(Foo7&& other) noexcept
        : d_allocator(other.d_allocator)
        , d_bar(nullptr, {d_allocator})
    {
        d_bar.reset(other.d_bar.release());
    }
//...
}
```

↓

```
## vector<Foo7> test
Allocating 24 bytes
Allocating 32 bytes
Allocating 48 bytes
Allocating 32 bytes
```

Save the cache

```
class polymorphic_allocator_delete {
  //...
  private:
    std::pmr::polymorphic_allocator<std::byte> d_allocator;
};

class Bar7 {
  //...
  private:
    std::pmr::string data;
};

class Foo7 {
  //...
  private:
    std::pmr::polymorphic_allocator<std::byte>          d_allocator;
    std::unique_ptr<Bar7, polymorphic_allocator_delete> d_bar;
};
```

```cpp
class Foo8 {
    Bar8                                       *d_bar;
    //^^^^
    //New, using a pointer instead of a unique_ptr.
    //...
  public:
    Foo8(std::pmr::polymorphic_allocator<std::byte> allocator = {})
    {
        std::pmr::polymorphic_allocator<Bar8> barAlloc{allocator};
        d_bar = barAlloc.allocate(1);
        try {
            barAlloc.construct(d_bar, allocator);
            //                 ^^^^^
            //New, we're allocating and constructing the pointer directly
        } catch(...) {
            alloc.deallocate(d_bar, 1);
            throw;
        }
    }
    //...

    ~Foo8() // <--- New, a custom destructor
    {
        if(d_bar) {
            std::pmr::polymorphic_allocator<Bar8> barAlloc = get_allocator();
            barAlloc.destroy(d_bar);
            barAlloc.deallocate(d_bar, 1);
        }
    }
    //...
};
```

```
## vector<Foo8> test
Allocating 16 bytes <- from 24
Allocating 32 bytes
Allocating 32 bytes <- from 48
Allocating 32 bytes
```

```
class Bar9 {
  public:
    //...
    std::pmr::polymorphic_allocator<std::byte> get_allocator()
        //                                     ^^^^^^^^^^^^^^^
        // New.
    {
        return data.get_allocator();
    }
  private:
    std::pmr::string data;
};

class Foo9 {
    //...
    // No more allocator member.
  public:
    //...
    std::pmr::polymorphic_allocator<std::byte> get_allocator()
        // Return the allocator this object was constructed with.
    {
        return d_bar->get_allocator();
    }
};


## vector<Foo9> test
Allocating 8 bytes  <- from 16
Allocating 32 bytes
Allocating 16 bytes <- from 32
Allocating 32 bytes
```

Allocator awareness best practices (1/2)

- Fix allocators at construction
- Allocator argument to constructor, copy constructor, move constructor, and move copy constructor (generally) passing allocator to data members
- "dirty little secret" move constructor
- `polymorphic_allocator<std::byte>` `allocator_type` member type

Allocator awareness best practices (2/2)

- `get_allocator` returns allocator passed in constructor
- Delegate to data members for allocator storage
- Always use global storage for the default allocator
- Set the default allocator only in main

Allocators and move semantics

```
std::pmr::vector<int> f() {
    std::pmr::vector<int> result(some_allocator);
    //...
    return result;
}

//...

std::pmr::vector<int> v = f();

std::pmr::vector<int> w;
w = f();
```

Pass in allocator of return value?

```
std::pmr::vector<int> f(std::pmr::polymorphic_allocator<std::byte>);

std::pmr::vector<int> v = f(some_allocator);

std::pmr::vector<int> w(some_allocator);
w = f(some_allocator);

std::pmr::vector<int> x;
x = f(some_allocator);
```

Pass in allocator of return value?

- Error prone
- Depends on "dirty little secret"
- Allocators involved in almost every function call

Pass by pointer/reference?

```
void f(std::pmr::vector<int>*);

std::pmr::vector<int> v;
f(&v);

std::pmr::vector<int> w(some_allocator);
f(&w);
```

Pass by pointer/reference?

- Two stage initialization
- Loose `const`

User cost

```cpp
std::pmr::string s = std::move(somedatatype.somedatamember);
```

```
class CoolThing {
    std::pmr::string d_buffer;
    std::pmr::vector<int> d_intbuffer;
    std::function<void (std::string)> d_sendBuffer;

public:
    void flush() {
        d_sendBuffer(std::move(d_buffer));
    }
}
```

```
class CoolThing {
    std::pmr::monotonic_buffer_resource d_resource; // Yo, so cool
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    std::pmr::string d_buffer;
    std::pmr::vector<int> d_intbuffer;
    std::function<void (std::string)> d_sendBuffer;

  public:
    void CoolThing() :
        d_resource( 1000 /* big buffer */,
                    std::pmr::get_default_resource() ),
        d_buffer({d_resource})
    {
    }

    void flush() {
        d_sendBuffer(std::move(d_buffer));
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ allocator leak!
    }
}
```

Development Cost

```cpp
struct Foo {
  int d_i = 0;
  std::vector<int> d_v{};
};
```

```cpp
struct Foo {
  using allocator_type = std::pmr::polymorphic_allocator<std::byte>;

  Foo(std::pmr::dynamic_allocator<std::byte> alloc = {})
      : d_v(alloc) {}

  Foo(int i, std::pmr::dynamic_allocator<std::byte> alloc = {})
      : d_i(i), d_v(alloc) {}

  Foo(int i, const std::pmr::vector<int> &v,
      std::pmr::dynamic_allocator<std::byte> alloc = {})
      : d_i(i), d_v(v, alloc) {}

  Foo(int i, std::pmr::vector<int> &&v,
      std::pmr::dynamic_allocator<std::byte> alloc = {})
      : d_i(i), d_v(std::move(v), alloc) {}

  Foo(const Foo &other,
      std::pmr::dynamic_allocator<std::byte> alloc = {})
      : d_i(other.d_i), d_v(other.d_v, alloc) {}

  Foo(Foo &&other) noexcept
      : d_i(other.d_i), d_v(std::move(other.d_v)) {}

  Foo(Foo &&other,
      std::pmr::dynamic_allocator<std::byte> alloc)
      : d_i(other.d_i), d_v(std::move(other.d_v), alloc) {}

  Foo& operator=(const Foo &other) {
    d_i = other.d_i;
    d_v = other.d_v;
    return *this;
  }

  Foo& operator=(Foo &&other) {
    d_i = other.d_i;
    d_v = std::move(other.d_v);
    return *this;
  }

  allocator_type get_allocator() const noexcept { return d_v.get_allocator(); }

  int d_i = 0;
  std::pmr::vector<int> d_v{};
};
```

Standard Support

In: containers, `string`, `tuple`

Out: `variant`, `function`

Drawback summary

- Loose return-by-value
- Loose single-step initialization
- Loose `const`
- Sometimes wasteful with memory
- Tricky exception safety issues
- Lost move-construction/move-assignment connection
- Lots of boilerplate
- Spotty standard support

Alternatives

- Object pools
- Custom data types

C++17's std::pmr Comes With a Cost

C++Now 2018

David Sankel
@david_sankel
Bloomberg