### ADD TEST POLYMORPHIC MEMORY RESOURCE TO THE STANDARD LIBRARY

**CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version History</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Motivation</td>
<td>3</td>
</tr>
<tr>
<td>Why is this appropriate for LEWG attention?</td>
<td>3</td>
</tr>
<tr>
<td>Testing Features Do Not Belong In The Standard</td>
<td>3</td>
</tr>
<tr>
<td>This Problem Is Better Addressed With External Tools</td>
<td>4</td>
</tr>
<tr>
<td>We Spend Too Much Time Talking About Allocators</td>
<td>4</td>
</tr>
<tr>
<td>Experience</td>
<td>4</td>
</tr>
<tr>
<td>Features</td>
<td>5</td>
</tr>
<tr>
<td>Examples of Use</td>
<td>5</td>
</tr>
<tr>
<td>Memory Leak Detection (stage1)</td>
<td>6</td>
</tr>
<tr>
<td>Wrong Alignment and Buffer Overrun Detection (stage2)</td>
<td>8</td>
</tr>
<tr>
<td>Wrong Number of Bytes in Deallocate (stage 3)</td>
<td>9</td>
</tr>
<tr>
<td>Success of Create/Destroy (stage4)</td>
<td>9</td>
</tr>
<tr>
<td>Deallocation of Already DEALLOCATED Pointer (stage4a)</td>
<td>10</td>
</tr>
<tr>
<td>Implemented a Copy Constructor (stage5)</td>
<td>11</td>
</tr>
<tr>
<td>Wrong Assignment Operator (stage6)</td>
<td>13</td>
</tr>
<tr>
<td>Implemented a Copy Assignment Operator (stage7)</td>
<td>15</td>
</tr>
</tbody>
</table>
The Proposed Entities in Alphabetical Order ................................................................. 20

Default Resource Guard ................................................................................................ 20

Exception Test Loop ..................................................................................................... 21

Design considerations .................................................................................................. 22

Test Resource .............................................................................................................. 23

Test Resource Exception ............................................................................................ 25

Test Resource Monitor ............................................................................................... 26

Proposed wording changes ........................................................................................ 27

Extend 19.12.1 Header <memory_resource> synopsis [mem.res.syn] .......................... 27

Add 19.12.5 Default resource guard [mem.res.defguard] ......................................... 28

Renumber 19.12.5, 19.12.6 ........................................................................................... 28

Insert section 19.12.8 Testing support [mem.res.test] ............................................. 29

Acknowledgements ..................................................................................................... 38

References .................................................................................................................. 38

VERSION HISTORY

R0: Initial draft

R1: Added new sections addressing LEWGI feedback on relevance of the subject matter
    Removed special handling of 0 sized allocations

INTRODUCTION

This document proposes adding to the C++ Standard Library an instrumented polymorphic memory resource (and its accompanying types) and an algorithm to support testing exception safety. The proposed test_resource implements the std::pmr::memory_resource abstract interface and can be used to track various details of memory allocated from it. Those available statistics include the number of outstanding allocated memory blocks (and bytes) that are currently in use, the cumulative number of blocks (and bytes) that have been allocated, and the maximum number of blocks (and bytes) that have been in use at any one time. The test_resource can also be configured to throw an exception after the number of allocation requests exceeds some specified limit,
thus enabling testing of exception safety in face of allocation failures using the `exception_test_loop` algorithm.

**MOTIVATION**

Testing is hard. Testing code that manages memory is harder. The polymorphic memory resource (together with the idea of the replaceable default resource) gives us the ability to replace (or extend) the memory resource with capabilities that allow testing and monitoring of memory management as well as testing robustness in face of memory allocation failure.

When testing a type (or template) that manages memory resources one would like to be assisted in finding hard bugs, such as memory leaks, double releases, use of already released memory, and exception safety issues when an allocation fails. It is also helpful to be able to monitor memory resource usage (or a change in it), such as if a type allocates memory using the default resources when it should not have.

Having a polymorphic memory resource that supports such testing needs allows the programmer to test (polymorphic) allocator enabled code without requiring external tools, or analyzing log files. Testing proper memory management (and robustness in face of memory allocation failure) can be made a normal part of test code. That, in turn, enables targeted testing of individual objects that use polymorphic memory resources, such as using separate `test_resource` s for different objects. What we propose also allows precise testing of local behavior in face of allocation failure. Such validation is hard (if not impossible) to achieve with external tools.

**WHY IS THIS APPROPRIATE FOR LEWG ATTENTION?**

This paper was presented to LEWGI at San Diego 2018, and given several important questions to answer before the group would consider this proposal on-topic.

**TESTING FEATURES DO NOT BELONG IN THE STANDARD**

First, the topic material is to support testing. There is a concern that test frameworks have no place in the standard, as there are a variety of successful frameworks already competing in the marketplace, and wg21 has no business approving only one.

However, the proposal under discussion is not a test framework, but rather a tool to support testing that can be applied within any testing framework. There is not even a prototype frame as part of the proposal. Wg21 has already clearly signaled that it does intend to support facilities that help with the full software lifecycle, by adopting `source_location` in C++20, approving a stack trace library for C++23 (once it passes LWG wording review), and instituting SG21 to finalize the work on a contracts facility, also targeting C++23.

However, this proposal is deliberately targeting Library Fundamentals 3 as the variety of information the proposed facility could choose to track is large, and finding the optimal feature set would require feedback from a wider user base with competing demands. See the Experience section below if still concerned that the lower bar for experience has not been achieved.

One additional concern raised was that if we have the facility already, why do we need to add it to the standard? Strictly speaking, we do not. We can happily use our own implementation, and make an open source version available. However, the ability to test correct usage of memory resources turns out to be a significant enough concern that the ability to test such usage has always been considered an integral part of the design. Standardizing some kind of testing resource facility would greatly help in the adoption of the existing memory
resource library. Running that exercise through a Library TS seems the best way to get the necessary feedback on whether full adoption by the standard itself is the right answer.

**THIS PROBLEM IS BETTER ADDRESSED WITH EXTERNAL TOOLS**

There was a suggestion that the market already demonstrated the benefit of external tools to instrument memory allocation, and any work in this space would better belong in SG15. It is true that global memory allocators can be well tracked by such external tooling. However, such tools do not address the needs of stateful allocators so easily. The concerns go beyond the global head, and into whether each object is allocating memory from the correct resource. Testing at this level is handled much more easily within a regular testing framework than with an external tool. Also, the proposed facility goes beyond simple testing of memory allocation, and into telemetry and a tool to test exception safety. While it may be possible to address these concerns with a high level of customization on external tools, it is nowhere near as simple as just plugging in an adapted memory resource that is fit for purpose.

**WE SPEND TOO MUCH TIME TALKING ABOUT ALLOCATORS**

There is a concern that every time we talk about allocators, we are further complicating an already complex part of the language, and we will lose too much time educating folks about the topic under discussion. This proposal is a pure extension, using only the simplest of extension hooks already provided by the library – we implement a class overriding the pure virtual functions in its base class. The intent when standardizing memory resources is that further resources could be easily added in the future, as they demonstrated their benefit. This proposal plugs directly into that features touching nothing else. Historically, it has also been the main application of the `default_resource` function that was adopted in C++17. This library was not proposed earlier as we provided the strictly minimal useful set of resources to bootstrap the library for C++, but is possibly the most widely used resource within our own code base.

**EXPERIENCE**

The proposed types are not experimental. Their roots can be traced back to the one originally conceived and developed by John Lakos at Bear Sterns (c. 1997) as part of his polymorphic memory allocator model, which itself evolved into the PRM facility now part of C++17. They have also been in use by Bloomberg LP for nearly two decades in testing various software components, including (but not limited to) our own Standard Library implementation. We propose adding those types into standard C++ with just changes in the naming convention as well as removing the use of macros in the automated allocation-failure testing.

The notable differences between the proposed facilities and Bloomberg LP’s solution are:

- The Bloomberg implementation uses macros (and not algorithms) to implement the test loop.
- Bloomberg’s polymorphic memory resource implementation (the `BloombergLP::bslma::Allocator` protocol), unlike the standard `pmr`, does not support alignment, and does not support a `size` parameter for the deallocate method.

Also note that while Bloomberg LP has other allocators used in testing (such as an allocator that supports functionality similar to Electric Fence https://en.wikipedia.org/wiki/Electric_Fence) that compose with the test memory resource; we are not proposing them for standardization at this time.
FEATURES

The proposed test_resource type (and its accompanying types and algorithm) provide the following features:

- a thread-safe implementation of the polymorphic memory resource interface
- the detection of memory leaks
- the detection of double releasing of memory
- detection of writing before or beyond the allocated memory area (boundary violation)
- overwriting memory just before deallocating it to help detect use of deleted memory
- tracking of memory use statistics (number of outstanding blocks and bytes) that are currently in use, the cumulative number of blocks (and bytes) that have been allocated, and the maximum number of blocks (and bytes) that have been in use at any one time
- monitoring of memory use changes (via the test_resource_monitor type)
- temporary replacement of the default memory resource using the default_resource_guard
- testing (exception safety) behavior in case of memory allocation failure (when the resource throws) using the test_allocation_failure algorithm

EXAMPLES OF USE

This section uses a primitive little string implementation as demonstration. The string is called pstring and is provided only for demonstration purposes. The examples of the code go in stages. As the pstring class is being built up, and its errors removed, we demonstrate different capabilities of the test_resource, starting with the detection of memory leaks. The exception test example uses standard pmr types. Note that in the listings that follow, the pstring class is being tested, implemented and enhanced in stages.

The complete source code of all the examples and a reference implementation of the proposed entities can be found in Bloomberg’s GitHub pages at https://github.com/bloomberg/p1160.
MEMORY LEAK DETECTION (STAGE1)

As you may see in Listing 1, the first “implementation” of `pstring` has several shortcomings. The most obvious is that it has no destructor so it leaks memory.

```cpp
Listing 1

class pstring {
   // This class is for demonstration purposes *only*.
public:
   using allocator_type = std::pmr::polymorphic_allocator<>
   pstring(const char *cstr, allocator_type allocator = {});
   allocator_type get_allocator() const {
      return m_allocator_;}
   std::string str() const { // For sanity checks only.
      return {m_buffer_, m_length_};}
private:
   allocator_type m_allocator_; 
   size_t m_length_; 
   char *m_buffer_;
};

inline
pstring::pstring(const char *cstr, allocator_type allocator)
   : m_allocator_(allocator)
   , m_length_(std::strlen(cstr))
   , m_buffer_(static_cast<char *>(m_allocator_.allocate_bytes(m_length_, 1)))
   {std::strcpy(m_buffer_, cstr);
```
Listing 2 shows the test code. Listing 3 shows the output of the `test_resource` reporting the memory leak. We identify the `test_resource` using the name `stage1`. While in this simple example the name does not matter, we may use several `test_resources`, for example an additional one for the default resource (to detect that it is not used). Listing 4 shows the `test_resource` output with the verbosity on. The verbose output may be used to debug memory management issues; although in this simple case it is not really necessary.

Listing 2

```cpp
std::pmr::test_resource tmr{ "stage1", verbose }; tmr.set_no_abort(true);

pstring astring{ "foobar", &tmr }; ASSERT_EQ(astring.str(), "foobar");
```

Listing 3

MEMORY_LEAK from stage1:
Number of blocks in use = 1
Number of bytes in use = 6

Listing 4

test_resource stage1 [0]: Allocated 6 bytes (aligned 1) at 00543F48.
==================================================
TEST RESOURCE stage1 STATE
--------------------------------------------------
Category        Blocks  Bytes
--------        ------  -----
IN USE        1       6
MAX        1       6
TOTAL        1       6
MISMATCHES        0
BOUNDS ERRORS        0
PARAM. ERRORS        0
--------------------------------------------------
Indices of Outstanding Memory Allocations: 0
MEMORY_LEAK from stage1:
Number of blocks in use = 1
Number of bytes in use = 6

Note that the actual format of the output is not specified by this proposal. The example output is what happens to be produced by the Bloomberg LP example implementation of the proposed features.
WRONG ALIGNMENT AND BUFFER OVERRUN DETECTION (STAGE2)

In Listing 5 we are adding a destructor to get rid of the memory leak and trigger the checks that are done during deallocation. As we are deallocating with the wrong alignment (alignment 2 instead of 1) we are getting an error message (from the test_resource) telling so. We have also allocated one-too-few bytes for the string (no space for the closing NULL character), so we are getting a buffer overrun error as well. See Listing 6. The memory leak is still reported because due to the errors (mismatch in the alignment on deallocate and the buffer overrun) the test_resource did not attempt to free the memory.

Listing 5

```cpp
inline
pstring::~pstring()
{
    m_allocator_.deallocate_bytes(m_buffer_, m_length_, 2);
}
```

Listing 6

```plaintext
*** Freeing segment at 00332CC8 using wrong alignment (2 vs. 1). ***
*** Memory corrupted at 1 bytes after 6 byte segment at 00332CC8. ***
Pad area after user segment:
00332CCE: 00 b1 b1 b1 b1 b1 b1 b1
Header:
00332CA0: ef be ad de 06 00 00 00 01 00 00 00 cd cd cd cd
00332CB0: 00 00 00 00 00 00 00 48 b6 32 00 d8 fc 23 00
00332CC0: b1 b1 b1 b1 b1 b1 b1 b1
User segment:
00332CC8: 66 6f 6f 62 61 72
MEMORY_LEAK from stage2:
    Number of blocks in use = 1
    Number of bytes in use = 1
```
WRONG NUMBER OF BYTES IN DEALLOCATE (STAGE 3)

We fix the allocation (to allocate enough space) and the alignment in the deallocation, but we “forget” to update the size (number of bytes) in the deallocation to match the allocation. See the changes to the code in Listing 7 and the resulting report from the test_resource in Listing 8.

Listing 7

```cpp
inline pstring::pstring(const char *cstr, allocator_type allocator)
 : m_allocator_(allocator),
   m_length_(std::strlen(cstr)),
   m_buffer_(m_allocator_.allocate_object<char>(m_length_ + 1)) {
   std::strcpy(m_buffer_, cstr);
}
inline pstring::~pstring() {
   m_allocator_.deallocate_object(m_buffer_, m_length_);
}
```

Listing 8

```text
*** Freeing segment at 003C2D48 using wrong size (6 vs. 7). ***
Header:
003C2D20:   ef be ad de 07 00 00 00 01 00 00 00 cd cd cd cd
003C2D30:   00 00 00 00 00 00 00 f8 b5 3b 00 a8 fc 22 00
003C2D40:   b1 b1 b1 b1 b1 b1 b1 b1 User segment:
003C2D48:   66 6f 6f 62 61 72 MEMORY_LEAK from stage3:
   Number of blocks in use = 1
   Number of bytes in use = 1
```

SUCCESS OF CREATE/DESTROY (STAGE4)

In Stage 4 we fix the deallocate call to use the proper byte size and the test code runs without any errors being reported.

Listing 9

```cpp
m_allocator_.deallocate_object(m_buffer_, m_length_ + 1);
```

Listing 10

```cpp
std::pmr::test_resource tpmr{ "stage4", verbose }; tpmr.set_no_abort(true);
pstring astring{ "foobar", &tpmr }; ASSERT_EQ(astring.str(), "foobar");```
DEALLOCATION OF ALREADY DEALLOCATED POINTER (STAGE4A)

In Stage 4a only the test code changes. We test the copy constructor of `pstring`. Since we have not declared a copy constructor, we have an implicitly defined one that leads to undefined behavior in the destructor because it does not deep copy the string. Listing 11 shows the new test code; Listing 12 shows the verbose output that indicates the error.

Listing 11
```
std::pmr::test_resource tmr{ "stage4a", verbose };  
tmr.set_no_abort(true);

pstring astring{ "foobar", &tmr };  
pstring string2{ astring };  

ASSERT_EQ(astring.str(), "foobar");  
ASSERT_EQ(string2.str(), "foobar");
```

Listing 12
```
test_resource stage4a [0]: Allocated 7 bytes (aligned 1) at 00815858.
test_resource stage4a [0]: Deallocated 7 bytes (aligned 1) at 00815858.  
*** Invalid magic number 0xdead0022 at address 00815858. ***
Header:
00815830:       22 00 ad de   07 00 00 00   01 00 00 00   00 00 00 00
00815840:       00 00 00 00   00 00 00 00   b0 d8 80 00   f8 f8 3b 00
00815850:       b1 b1 b1 b1   b1 b1 b1 b1
User segment:
00815858:       a5 a5 a5 a5   a5 a5 a5 a5
===================================================================
TEST RESOURCE stage4a STATE
--------------------------------------------------
Category        Blocks  Bytes
--------        ------  ----- 
IN USE        0       0
MAX              1       7
TOTAL          1       7
MISMATCHES      1
BOUNDS ERRORS   0
PARAM. ERRORS   0
```
IMPLEMENTED A COPY CONSTRUCTOR (STAGE5)

In the Stage 5 version of `pstring` we have implemented a copy constructor, as seen in Listing 13. The revised test code is in Listing 14. The verbose output of running the test is in Listing 15.

Listing 13

```cpp
    pstring(const pstring& other, allocator_type allocator = {});

    // Additional members omitted for brevity

inline
    pstring::pstring(const pstring& other, allocator_type allocator)
        : m_allocator_(allocator),
          m_length_(other.m_length_),
          m_buffer_(m_allocator_.allocate_object<char>(m_length_ + 1))
    {
        std::strcpy(m_buffer_, other.m_buffer_);
    }
```

Listing 14

```cpp
    std::pmr::test_resource dpmr{ "default", verbose };  
    std::pmr::default_resource_guard dg(&dpmr);

    std::pmr::test_resource tpmr{ "stage5", verbose };  
    tpmr.set_no_abort(true);

    pstring astring{ "foobar", &tpmr };     
    pstring string2{ astring };      

    ASSERT_EQ(astring.str(), "foobar");  
    ASSERT_EQ(string2.str(), "foobar");
```

As `string2` uses the default resource we use a `default_resource_guard` (also introduced in this proposal) to test its memory management activities.
test_resource stage5 [0]: Allocated 7 bytes (aligned 1) at 00715858.
test_resource default [0]: Allocated 7 bytes (aligned 1) at 00715898.
test_resource default [0]: Deallocated 7 bytes (aligned 1) at 00715898.
test_resource stage5 [0]: Deallocated 7 bytes (aligned 1) at 00715858.

Test Resource stage5 State

<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN USE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>BOUNDS ERRORS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PARAM. ERRORS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Test Resource default State

<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN USE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>BOUNDS ERRORS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PARAM. ERRORS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
WRONG ASSIGNMENT OPERATOR (STAGE6)

In this stage we are testing a wrong copy assignment operator. Note that due to the use of `polymorphic_allocator<>` (as a member) the compiler does not generate a default copy assignment operator, so we have to implement a wrong one by hand. See Listing 17. Listing 18 is the verbose output showing double release of the memory that is a result of memberwise copy assignment.

**Listing 16**

```cpp
std::pmr::test_resource tpmr{ "stage5a", verbose };  
tpmr.set_no_abort(true);

pstring astring{ "foobar", &tpmr };  
pstring string2{ "string", &tpmr };

string2 = astring;

ASSERT_EQ(astring.str(), "foobar");  
ASSERT_EQ(string2.str(), "foobar");
```

**Listing 17**

```cpp
inline pstring& pstring::operator=(const pstring& rhs)
{
  m_length_ = rhs.m_length_;  
  m_buffer_ = rhs.m_buffer_;  
  return *this;
}
```
Listing 18

test_resource stage5a [0]: Allocated 7 bytes (aligned 1) at 00455858.
test_resource stage5a [1]: Allocated 7 bytes (aligned 1) at 004558D8.
test_resource stage5a [0]: Deallocated 7 bytes (aligned 1) at 00455858.
*** Invalid magic number 0xdead0032 at address 00455858. ***

Header:
00455830: 32 00 ad de 07 00 00 00 01 00 00 00 00 00 00 00
00455840: 00 00 00 00 00 00 00 00 c8 d8 44 00 58 f8 3c 00
00455850: b1 b1 b1 b1 b1 b1 b1 b1

User segment:
00455858: a5 a5 a5 a5 a5 a5 a5 a5

=====================================================================
TEST RESOURCE stage5a STATE
=====================================================================

<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>----</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>IN USE</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>MAX</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BOUNDS ERRORS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PARAM. ERRORS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Indices of Outstanding Memory Allocations:
1

MEMORY_LEAK from stage5a:
Number of blocks in use = 1
Number of bytes in use = 1
IMPLEMENTED A COPY ASSIGNMENT OPERATOR (STAGE7)

In this stage we implement a copy assignment operator as seen in Listing 19. The verbose test output is in Listing 20. The test code is the same as Stage 6. Note that this assignment operator will still fail in case of self-assignment, as seen in the next stage.

Listing 19

```cpp
inline
pstring& pstring::operator=(const pstring& rhs)
{
    char *buff = m_allocator_.allocate_object<char>(rhs.m_length_ + 1);
    m_allocator_.deallocate_object(m_buffer_, m_length_ + 1);
    m_buffer_ = buff;
    std::strncpy(m_buffer_, rhs.m_buffer_);
    m_length_ = rhs.m_length_;

    return *this;
}
```

Listing 20

```
test_resource stage7 [0]: Allocated 7 bytes (aligned 1) at 003A5858.
test_resource stage7 [1]: Allocated 7 bytes (aligned 1) at 003A5958.
test_resource stage7 [2]: Allocated 7 bytes (aligned 1) at 003A5918.
test_resource stage7 [1]: Deallocated 7 bytes (aligned 1) at 003A5958.
test_resource stage7 [2]: Deallocated 7 bytes (aligned 1) at 003A5918.
test_resource stage7 [0]: Deallocated 7 bytes (aligned 1) at 003A5858.
```

```
-------------------------------------------------------------------
TEST RESOURCE stage7 STATE
-------------------------------------------------------------------
Category        Blocks  Bytes
--------        ------  -----  
IN USE        0       0
MAX        3       21
TOTAL        3       21
MISMATCHES        0
BOUNDS ERRORS        0
PARAM. ERRORS        0
-------------------------------------------------------------------
```
SELF-ASSIGNMENT TEST (STAGE7A)

Self-assignment is not handled properly in the copy assignment operator code so this test will use deallocated memory (overwritten by the `test_resource` before deallocation) and therefore fail.

Listing 21

```cpp
std::pmr::test_resource tpmr{ "stage6a", verbose };  
tpmr.set_no_abort(true);

pstring astring{ "foobar", &tpmr };

astring = astring;

ASSERT_EQ(astring.str(), "foobar");
```

Listing 22

```
test_resource stage7a [0]: Allocated 7 bytes (aligned 1) at 00575918.
test_resource stage7a [1]: Allocated 7 bytes (aligned 1) at 00575858.
test_resource stage7a [0]: Deallocated 7 bytes (aligned 1) at 00575918.
astring.str() != "foobar"
NNNNNN != foobar
```

```
test_resource stage7a [1]: Deallocated 7 bytes (aligned 1) at 00575858.
```

`---
TEST RESOURCE stage7a STATE
---`

```
<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN USE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BOUNDS ERRORS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PARAM. ERRORS</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

---
SELF-ASSIGNMENT FIXED (STAGE 8)

In this last stage of `pstring` development we avoid self-assignment with an early return. The test code is the same as in Stage 7a. The copy-assignment operator code is changed as seen in Listing 23. The verbose output in Listing 24 shows that the copying did not happen (there is only one allocation/deallocation pair).

Listing 23

```cpp
inline
pstring& pstring::operator=(const pstring& rhs)
{
    if (this == &rhs) {
        return *this; // RETURN
    }
    char *buff = m_allocator_.allocate_object<char>(rhs.m_length_ + 1);
    m_allocator_.deallocate_object(m_buffer_, m_length_ + 1);
    m_buffer_ = buff;
    std::strcpy(m_buffer_, rhs.m_buffer_);
    m_length_ = rhs.m_length_;
    return *this;
}
```

Listing 24

test_resource stage8 [0]: Allocated 7 bytes (aligned 1) at 00425858.
test_resource stage8 [0]: Deallocated 7 bytes (aligned 1) at 00425858.

===================================================================
TEST RESOURCE stage8 STATE
===================================================================

<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN USE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Bounds Errors</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Param. Errors</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

===================================================================
TESTING ROBUSTNESS AGAINST ALLOCATION FAILURE

Testing classes that manage elements that allocate memory is difficult. We have to verify that if any of the allocations fail, there are no memory leaks or other mismanagement of resources. Without stateful allocators that task would be daunting, but with polymorphic memory resources and the test_resource it is quite easy.

The <test_resource> header provides the std::pmr::exception_test_loop algorithm that uses a std::pmr::test_resource to make sure that every allocation done by the tested code fails with an exception once. This is done by using the allocation limit of the test_resource in a loop. We start with a 0 allocation limit and we gradually increase it (in a loop) until we get no more test exceptions thrown by allocations. At that time the test succeeded. See Listing 25 for the example code and Listing 26 for the verbose output.

Listing 25

```cpp
std::pmr::test_resource tpmr{ "exception_tester", verbose };  
const char *longstr = "A very very long string that allocates memory";

std::pmr::exception_test_loop(tpmr,  
  [longstr](std::pmr::memory_resource *pmrp) {  
    std::pmr::deque<std::pmr::string> deq{ pmrp };  
    deq.emplace_back(longstr);  
    deq.emplace_back(longstr);  
    ASSERT_EQ(deq.size(), 2);  
  });
```
In the verbose output (Listing 26) one can observe the operation of the test loop. First, an immediate allocation failure. Then one allocation succeeds, and because an exception is thrown it is also deallocated. Then we see the exception_test_loop catching the exception. The process continues until all 4 allocations succeed.

```
Listing 26

*** exception: alloc limit = 0, last alloc size = 28, align = 4 ***
test_resource tester [1]: Allocated 28 bytes (aligned 4) at 00641018.
test_resource tester [1]: Deallocated 28 bytes (aligned 4) at 00641018.

*** exception: alloc limit = 1, last alloc size = 48, align = 1 ***
test_resource tester [3]: Allocated 28 bytes (aligned 4) at 00641018.
test_resource tester [4]: Allocated 48 bytes (aligned 1) at 00641090.
test_resource tester [4]: Deallocated 48 bytes (aligned 1) at 00641090.
test_resource tester [3]: Deallocated 28 bytes (aligned 4) at 00641018.

*** exception: alloc limit = 2, last alloc size = 56, align = 4 ***
test_resource tester [6]: Allocated 28 bytes (aligned 4) at 00641018.
test_resource tester [8]: Allocated 56 bytes (aligned 4) at 00641120.
test_resource tester [8]: Deallocated 56 bytes (aligned 4) at 00641120.
test_resource tester [7]: Deallocated 28 bytes (aligned 4) at 00641018.

*** exception: alloc limit = 3, last alloc size = 48, align = 1 ***
test_resource tester [10]: Allocated 28 bytes (aligned 4) at 00641018.
test_resource tester [11]: Allocated 48 bytes (aligned 1) at 00641090.
test_resource tester [12]: Allocated 56 bytes (aligned 4) at 00641120.
test_resource tester [13]: Allocated 48 bytes (aligned 1) at 00644030.
test_resource tester [10]: Deallocated 28 bytes (aligned 4) at 00641018.
test_resource tester [11]: Deallocated 48 bytes (aligned 1) at 00641090.
test_resource tester [13]: Deallocated 48 bytes (aligned 1) at 00644030.
test_resource tester [12]: Deallocated 56 bytes (aligned 4) at 00641120.

=================================================================
TEST RESOURCE tester STATE
=================================================================

<table>
<thead>
<tr>
<th>Category</th>
<th>Blocks</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN USE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>4</td>
<td>180</td>
</tr>
<tr>
<td>TOTAL</td>
<td>10</td>
<td>416</td>
</tr>
<tr>
<td>MISMATCHES</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>BOUNDS ERRORS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PARAM. ERRORS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
THE PROPOSED ENTITIES IN ALPHABETICAL ORDER

In this section we are introducing the proposed elements in detail.

DEFAULT RESOURCE GUARD

The default resource guard is a simple RAII class that supports installing a new default polymorphic memory resource and then restoring of the original default polymorphic memory resource in its destructor. A possible implementation is in Listing 27. Its typical use is very simple, and in the context of this proposal it usually involves testing the use of the default allocator, like ensuring that an action that should not use the default allocator really does not use it. See

Listing 27

```cpp
namespace std::pmr {

class [[maybe_unused]] default_resource_guard {

    memory_resource * _OldResource;

public:
    explicit default_resource_guard(::std::pmr::memory_resource * _NewDefault) {
        assert(_NewDefault != nullptr);
        _OldResource = ::std::pmr::set_default_resource(_NewDefault);
    }

    default_resource_guard(const default_resource_guard&) = delete;
    default_resource_guard& operator=(const default_resource_guard&) = delete;

    ~default_resource_guard() {
        ::std::pmr::set_default_resource(_OldResource);
    }
};
}
```

Listing 28

```cpp
void default_resource_use_testing() {
    std::pmr::test_resource tr{ "object" }; 
    std::pmr::string astring{
        "A very very long string that will hopefully allocate memory",
        &tr }
    std::pmr::test_resource dr{ "default" }; 
    std::pmr::test_resource_monitor drm{ &dr }
    {
        std::pmr::default_resource_guard drg{ &dr };

        std::pmr::string string2{ astring, &tr };
    }

    ASSERT(drm.is_total_same());
}
```
EXCEPTION TEST LOOP

Validating the exception safety guarantees of an operation can be tedious without an automated method. Common vocabulary for exception safety guarantees is described by David Abrahams in [Abrahams, D. (2000). Exception-safety in Generic Components], where he also describes an automatic testing method to validate behavior in face of exceptions in section 7 (Automated testing for exception-safety). The exception-testing algorithm used in this proposal was developed independently by John Lakos at Bloomberg LP (c. 2002).

We propose exception_test_loop as a similar algorithm that takes advantage of the test_resource having a configurable limit to the number of allocations before it fails by throwing an exception. This algorithm runs a code block (e.g., a lambda, a functor, or a function pointer) in a loop, assuming memory is allocated by a supplied test_resource. In the first iteration we set the allocation limit of the test_resource to zero (0), which will cause the very first allocation to immediately fail (by throwing a Test Resource Exception, see Listing 32). If a test_resource_exception is thrown from the code block, the algorithm increases the allocation limit of the test_resource by one (1) and then repeats the loop. The loop ends when no test_resource_exception is thrown.

As long as all relevant allocations inside the code block use the provided test_resource this simple algorithm ensures that all relevant allocations in that code block will fail at least once (with an exception that inherits from std::bad_alloc) therefore ensuring that all failure code-paths are tested for leaks and other anomalies.

Listing 29 shows a possible implementation of this algorithm, while Listing 30 shows possible use to test deque.

Listing 29

```cpp
template <class CODE_BLOCK>
void exception_test_loop(test_resource& pmrp, CODE_BLOCK codeBlock) {
    for (long long exceptionCounter = 0; true; ++exceptionCounter) {
        try {
            pmrp.set_allocation_limit(exceptionCounter);
            codeBlock(pmrp);
            pmrp.set_allocation_limit(-1);
            return;
        } catch (const test_resource_exception& e) {
            if (e.originating_resource() != &pmrp) {
                printf("\t*** test_resource_exception: limit = %lld, "
                      "from unexpected test_resource %p, \n                      e.originating_resource(),
                      e.originating_resource()->name().length()
                      e.originating_resource()->name().data());
                throw;
            }
            else if (pmrp.is_verbose()) {
                printf("\t*** test_resource_exception: limit = %lld, "
                       "last size = %zu, align = %zu ***\n",
                       exceptionCounter,
                       e.bytes(),
                       e.alignment());
            }
        }
    }
}
```
Concerns to be aware of include:

- If `codeBlock` throws the expected `test_resource_exception` directly then the loop may repeat infinitely.
- If `codeBlock` handles `bad_alloc` exceptions and does not rethrow then subsequent failure paths will not be tested.
- The `codeBlock` should not manipulate external state that would affect subsequent iterations of the loop.

Listing 30

```cpp
std::pmr::test_resource tmr{ "tester" };
const char *longstr = "A very very long string that hopefully allocates memory";

std::pmr::exception_test_loop(tmr, [longstr](std::pmr::memory_resource& pmr) {
  std::pmr::deque<std::pmr::string> deq{ &pmr };
  deq.push_back(longstr);
  deq.push_back(longstr);

  ASSERT_EQ(deq.size(), 2);
});
```

**DESIGN CONSIDERATIONS**

The algorithm unconditionally changes the allocation limit of the supplied `test_resource`. This directly follows Bloomberg’s current experience with our macro based implementation. It is consistent with how allocators are used in our test drivers. We have considered restoring the specific allocation limit on successful completion of the loop, but that raises the question of what to do if exiting the loop successfully requires a higher allocation limit than the one set on the supplied `test_resource`.

The algorithm loses information about the number of allocations necessary to successfully complete operation of the code block. We have considered returning that number (`exceptionCounter`) but we are concerned that such a return value might be interpreted as an error code while both zero and non-zero values are potentially correct.

Another concern is that there is no validation of basic or strong exception safety guarantees after a test exception is caught. We have considered supplying an additional validation block (that could contain the relevant assertions) but have no clear experience with such an API and in particular how to communicate information between the tested code block and the validator. We are looking into this as a future extension with additional overloads.
TEST RESOURCE

The test_resource is a thread-safe, instrumented memory resource that implements the standard std::pmr::memory_resource abstract interface and can be used to track various aspects of memory allocated from it, in addition to automatically detecting a number of memory management violations that might otherwise go unnoticed.

The available features are:

- optionally specify a name for the test_resource that will be printed in reports
- optionally specify a backing memory resource
- detection (or assisting in detection) of memory management violations
  - memory leaks
  - deallocating already deallocated memory
  - buffer underruns
  - buffer overruns
- conditionally aborting on detected memory management violations
- conditionally printing about detected memory management violations
- conditionally printing about allocations/deallocations
- failing allocation after a set allocation limit is reached (if set)
- provide statistics
  - total number of allocations
  - total number of deallocations
  - total number of mismatched allocations (memory is not from this resource)
  - total number of bounds errors (underrun plus overrun)
  - total number of bad deallocate parameters (mismatch on size in bytes or alignment number)
  - current number of memory blocks in use
  - current number of bytes in use
  - maximum allocated number of memory blocks any given time
  - maximum allocated number of bytes any given time
  - total number of blocks allocated
  - total number of bytes allocated
  - last allocated number of bytes
  - last deallocated number of bytes
  - last allocated address
  - last deallocated address

The interface of the test_resource can be divided into the following sections:

- constructors/destructors
- the implementation of the std::pmr::memory_resource interface
- the control and access interface for the settings
- access to the instrumentation values
- the status and the print function
namespace std::pmr {

class test_resource : public memory_resource {
public:
    test_resource(const test_resource&) = delete;
    test_resource& operator=(const test_resource&) = delete;
    test_resource();
    explicit test_resource(memory_resource *pmrp);
    explicit test_resource(const char *name);
    explicit test_resource(bool verbose);
    test_resource(bool verbose, memory_resource *pmrp);
    test_resource(const char *name, bool verbose);
    test_resource(const char *name, bool verbose, memory_resource *pmrp);
    ~test_resource();

    [[nodiscard]] void *do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void *p, size_t bytes, size_t alignment) override;
    bool do_is_equal(const memory_resource& that) const noexcept override;

    void set_allocation_limit(long long limit) noexcept;
    void set_no_abort(bool is_no_abort) noexcept;
    void set_quiet(bool is_quiet) noexcept;
    void set_verbose(bool is_verbose) noexcept;
    long long allocation_limit() const noexcept;
    bool is_no_abort() const noexcept;
    bool is_quiet() const noexcept;
    bool is_verbose() const noexcept;
    const char *name() const noexcept;
    void *last_allocated_address() const noexcept;
    size_t last_allocated_bytes() const noexcept;
    void *last_deallocated_address() const noexcept;
    size_t last_deallocated_bytes() const noexcept;

    long long allocations() const noexcept;
    long long blocks_in_use() const noexcept;
    long long max_blocks() const noexcept;
    long long total_blocks() const noexcept;
    long long bounds_errors() const noexcept;
    long long bad_deallocate_params() const noexcept;
    long long bytes_in_use() const noexcept;
    long long max_bytes() const noexcept;
    long long total_bytes() const noexcept;
    long long deallocations() const noexcept;
    long long mismatches() const noexcept;

    void print() const noexcept;
    long long status() const noexcept;
};
} // close namespace

TEST RESOURCE EXCEPTION

The `test_resource_exception` is thrown by the `test_resource` when its allocation limit is reached and there is an attempt to allocate further memory. It is basically a special form of `std::bad_alloc` that allows the exception tester algorithm to differentiate between real out-of-memory situations from the test-induced limits.

The exception inherits from `std::bad_alloc` so that when it is thrown, the same code path will be traveled like in case of a real allocation failure. In other words: to ensure that we test the code that would run in production, in case `std::bad_alloc` is thrown by a memory resource.

Listing 32

```cpp
class test_resource_exception : public :std::bad_alloc {
    test_resource *m_originating_;
    size_t m_size_;  
    size_t m_alignment_;  

    public:
    explicit test_resource_exception(
        test_resource *originating,
        size_t size,
        size_t alignment)
        : m_originating_(originating)
        , m_size_(size)
        , m_alignment_(alignment)
    {
    
    }

    const char *what() const noexcept override {
        return "std::pmr::test_resource_exception";
    }

    test_resource *originating_resource() const noexcept {
        return m_originating_;}

    size_t size() const noexcept {
        return m_size_;}

    size_t alignment() const noexcept {
        return m_alignment_;}

};
```
The `test_resource_monitor` works in tandem with `test_resource` to observe changes (or lack of changes) in the statistics collected by a `test_resource`. Note that the monitored statistics are based on number of memory blocks and do not depend on the number of bytes in those allocated blocks.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Same</th>
<th>up</th>
<th>down</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks_in_use</td>
<td>is_in_use_same</td>
<td>is_in_use_up</td>
<td>is_in_use_down</td>
</tr>
<tr>
<td>max_blocks</td>
<td>is_max_same</td>
<td>is_max_up</td>
<td>none</td>
</tr>
<tr>
<td>total_blocks</td>
<td>is_total_same</td>
<td>is_total_up</td>
<td>none</td>
</tr>
</tbody>
</table>

```cpp
class test_resource_monitor {
    public:
        explicit test_resource_monitor(const test_resource *monitored) noexcept;
        test_resource_monitor(const test_resource_monitor&) = delete;
        test_resource_monitor& operator=(const test_resource_monitor&) = delete;

        void reset() noexcept;

        bool is_in_use_down() const noexcept;
        bool is_in_use_same() const noexcept;
        bool is_in_use_up() const noexcept;
        bool is_max_same() const noexcept;
        bool is_max_up() const noexcept;
        bool is_total_same() const noexcept;
        bool is_total_up() const noexcept;

        long long delta_blocks_in_use () const noexcept;
        long long delta_max_blocks() const noexcept;
        long long delta_total_blocks () const noexcept;
};
```
The proposed wording changes are all additions, except for section renumbering, relative to N4762.

EXTEND 19.12.1 HEADER <MEMORYRESOURCE> SYNOPSIS [MEM.RES.SYN]

Additions are marked with brown background. Changes are underlined (just section renumbering).

namespace std::pmr {
// 19.12.2, class memory_resource
class memory_resource;

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;
bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;

// 19.12.3, class template polymorphic_allocator
template<class Tp> class polymorphic_allocator;

template<class T1, class T2>
bool operator==(const polymorphic_allocator& a, const polymorphic_allocator& b) noexcept;
template<class T1, class T2>
bool operator!=(const polymorphic_allocator& a, const polymorphic_allocator& b) noexcept;

// 19.12.4, global memory resources
memory_resource* new_delete_resource() noexcept;
memory_resource* null_memory_resource() noexcept;
memory_resource* set_default_resource(memory_resource* r) noexcept;
memory_resource* get_default_resource() noexcept;

// 19.12.5, class default_resource_guard
class default_resource_guard;

// 19.12.6, pool resource classes
struct pool_options;
class synchronized_pool_resource;
class unsynchronized_pool_resource;

// 19.12.7 class monotonic_buffer_resource
class monotonic_buffer_resource;

// 19.12.8 testing support
class test_resource;
class test_resource_exception;
class test_resource_monitor;

template <class Tp>
void exception_test_loop(test_resource& tr, Tp code);
}
ADD 19.12.5 DEFAULT RESOURCE GUARD [MEM.RES.DEFGUARD]

19.12.5 Class default_resource_guard [mem.res.defguard]

namespace std::pmr {
  class default_resource_guard {
    public:
      explicit default_resource_guard(memory_resource* r);
      ~default_resource_guard();

      default_resource_guard(const default_resource_guard&) = delete;
      default_resource_guard& operator=(const default_resource_guard&) = delete;
    private:
      memory_resource *old_default; // exposition only
  };
}

1 An object of type default_resource_guard controls the setting of the default memory resource within a block ([stmt.block]). A memory_resource is given to the guard at construction, which the guard sets as the default resource (as if by calling set_default_resource) and saves the previously set default resource. The guard object restores the previous default resource (using set_default_resource) when destroyed. The behavior of the program is undefined if the supplied memory_resource is destroyed before the default_resource_guard object.

explicit default_resource_guard(memory_resource* r);

2 Requires: r is not nullptr.

3 Effects: As if by old_default = set_default_resource(r);

4 Postconditions: get_default_resource() == r

5 Throws: Nothing.

~default_resource_guard();

6 Effects: As if by set_default_resource(old_default);

7 Postconditions: get_default_resource() == old_default

RENUMER 19.12.5, 19.12.6

19.12.8 Testing support

Testing support provides several types and an algorithm to aid in testing memory handling of types using polymorphic memory resources.

All attempts to catch allocator misuse are necessarily imprecise as any such issue is undefined behavior or an out-of-contract call. [Note: The program may abort or fail catastrophically in other ways, too. – End Note] False negatives (missed detections) are permitted for any specification below that mandates detecting an error. False positives are never allowed.

19.12.8.1 Class test_resource

namespace std::pmr {
    class test_resource : public memory_resource {
        [[nodiscard]] 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& that) const noexcept override;
    public:
        test_resource() : test_resource(false, "", new_delete_resource()) {}  
        explicit test_resource(memory_resource* upstream) : test_resource(false, ",", upstream) {} 
        explicit test_resource(string_view name) : test_resource(false, name, new_delete_resource()) {}  
        explicit test_resource(bool verbose) : test_resource(verbsole, ",", new_delete_resource()) {}  
        test_resource(string_view name, memory_resource* upstream) : test_resource(false, name, upstream) {}  
        test_resource(bool verbose, memory_resource* upstream) : test_resource(verbsole, ",", upstream) {}  
        test_resource(bool verbose, string_view name) : test_resource(verbsole, name, new_delete_resource()) {} 
        test_resource(bool verbose, string_view name, memory_resource* upstream) 
        ~test_resource();
        test_resource(const test_resource& ) = delete;
        test_resource& operator=(const test_resource& ) = delete;

        void set_allocation_limit(long long limit) noexcept;
        void set_no_abort(bool flag) noexcept;
        void set_quiet(bool flag) noexcept;
        void set_verbose(bool flag) noexcept;
        long long allocation_limit() const noexcept;
        long long deallocations() const noexcept;
        long long allocations() const noexcept;
        long long total_blocks() const noexcept;
    }
};
long long bounds_errors() const noexcept;
long long bad_deallocate_params() const noexcept;
long long mismatches() const noexcept;
long long bytes_in_use() const noexcept;
long long bytes_max() const noexcept;
long long bytes_total() const noexcept;
void print() const noexcept;
bool has_errors() const noexcept;
bool has_allocations() const noexcept;
long long status() const noexcept;

private:
unspecified-integer allocations_;     // exposition only
unspecified-integer allocation_limit_; // exposition only
unspecified-integer deallocations_;   // exposition only
unspecified-integer blocks_in_use_;   // exposition only
unspecified-integer total_blocks_;     // exposition only
unspecified-integer max_blocks_;       // exposition only
unspecified-integer bytes_in_use_;     // exposition only
unspecified-integer total_bytes_;      // exposition only
unspecified-integer max_bytes_;        // exposition only
unspecified-integer last_deallocated_bytes_; // exposition only
unspecified-integer last_deallocated_alignment_; // exposition only
void *last_allocated_address_;        // exposition only
};

void test_resource(bool verbose, string_view name, memory_resource* upstream);

1 Requires: upstream != nullptr

2 Effects: Create a test_resource object that uses the specified upstream memory_resource. Optionally specify verbosity setting and a name. Initialize the remaining settings and instrumentation (as described in Postconditions). [Note: To avoid memory allocation the name is stored as a string_view, not deep copied. – End note] The behavior of the program is undefined if the supplied upstream objects is destroyed before the test_resource object.
3 Postconditions:
   allocation_limit() == -1
   is_no_abort() == false
   is_quiet() == false
   is VERBOSE() == verbose
   name() == name
   upstream_resource() == upstream
   last_allocated_address() == nullptr
   last_allocated_bytes() == 0
   last_allocated_alignment() == 0
   last_deallocated_address() == nullptr
   last_deallocated_bytes() == 0
   last_deallocated_alignment() == 0
   allocations() == 0
   deallocations() == 0
   blocks_in_use() == 0
   blocks_max() == 0
   blocks_total() == 0
   bounds_errors() == 0
   bad_deallocate_params() == 0
   bytes_in_use() == 0
   bytes_max() == 0
   bytes_total() == 0
   status() == 0

4 Throws: Nothing
~test_resource();

5 Effects: if is_VERBOSE() == true, call print. If is_quiet() == false check for and report allocations through this test memory resource that have not been deallocated by printing to the standard output. If such leaks are found and if is_no_abort() == true call abort().

   void* do_allocate(size_t bytes, size_t alignment) override;

6 Requires: alignment != 0

7 Effects: Increment allocations_. If allocation_limit_ is non-negative, decrement allocation_limit_ and if the limit becomes negative, throw a test_resource_exception with the supplied bytes and alignment. Otherwise allocate at least bytes using the upstream memory resource. [Note: Additional memory (larger than bytes) may be allocated to accommodate for buffer overrun/underrun verification and a memory-block descriptor that aids in identifying blocks allocated by a test_resource. – End note] Increment blocks_in_use_ and total_blocks_. max_blocks_ = max(max_blocks_, blocks_in_use_). Increase bytes_in_use_ and total_bytes_by bytes. max_bytes_ = max(max_bytes_, bytes_in_use_). Store the returned address into last_allocated_address_. If is_VERBOSE() == true print out, to the standard output, information about the allocation.

8 Postconditions:
   - last_allocated_bytes() == bytes
   - last_allocated_alignment() == alignment
9 Returns: a pointer well-aligned for alignment and pointing to at least bytes bytes of memory provided by the upstream memory resource

10 Throws: test_resource_exception or any exception thrown by the upstream memory resource void do_deallocate(void* ptr, size_t bytes, size_t alignment) override;

11 Effects: Increment deallocations_. A parameter error is detected if ptr != nullptr and the parameters do not match the bytes and alignment parameters provided to do_allocate(). A mismatched deallocation is detected if ptr != nullptr and this memory resource did not allocate that pointer. Underrun and overrun errors may be detected if guard bytes in either side of the allocated block do not have their expected values.

12 If any error is detected, increment its corresponding counter; if is_quiet() == true immediately return to the caller, otherwise report the errors found and is_no_abort() == false return immediately to the caller, otherwise call abort.

13 If no errors were detected update last_deallocated_bytes_ to bytes, last_deallocated_alignment_ to alignment, last_deallocated_address_ to ptr, decrement blocks_in_use_, and decrease bytes_in_use_ by bytes. Finally deallocate the memory block using the upstream memory resource. [Note: Implementations may overwrite the memory block before deallocation with a pattern that indicates deleted memory or use other tactics to detect use of deleted memory. – End note] If is_verbos() == true print out the details of the deallocation to the standard output.

bool do_is_equal(const memory_resource& that) const noexcept override;

14 Returns: this == &that;

void set_allocation_limit(long long limit) noexcept;

15 Effects: Sets the allocation limit to the supplied limit. [Note: Any negative value for limit means there is no allocation limit imposed by this test memory resource. – End Note]

16 Postconditions: allocation_limit() == limit

void set_no_abort(bool flag) noexcept;

17 Effects: Set the no-abort behavior. [Note: If flag is true, do not abort the program upon detecting errors. The default value of the setting is false. – End Note]

18 Postconditions: is_no_abort() == flag

void set_quiet(bool flag) noexcept;

19 Effects: Set the quiet behavior. [Note: If flag is true, do not report detected errors and imply is_no_abort() == true. The default value of the setting is false. – End Note]

20 Postconditions: is_quiet() == flag
void set_verbose(bool flag) noexcept;

21 Effects: Set the verbose behavior. [Note: If flag is true, report all allocations and deallocations to the standard output. The default value of the setting is false or what is specified in the constructor. – End Note]

22 Postconditions: isVerbose() == flag

long long allocation_limit() const noexcept;

23 Returns: the number of allocation requests permitted before throwing test_resource_exception or a negative value if this test memory resource does not impose a limit on the number of allocations [Note: This value will decrement with every call to do_allocate. – End Note]

bool is_no_abort() const noexcept;

24 Returns: the current no-abort flag

bool is_quiet() const noexcept;

25 Returns: the current quiet flag

bool is_verbose() const noexcept;

26 Returns: the current verbosity flag

string_view name() const noexcept;

27 Returns: the name supplied to this test_resource at construction

memory_resource* upstream_resource() const noexcept;

28 Returns: the pointer to the upstream memory_resource supplied to this test_resource at construction

void *last_allocated_address() const noexcept;

29 Returns: the pointer to the last memory block successfully allocated by this test_resource

size_t last_allocated_bytes() const noexcept;

30 Returns: the requested number of bytes of the last memory block successfully allocated by this test_resource

size_t last_allocated_alignment() const noexcept;

31 Returns: the requested alignment of the last memory block successfully allocated by this test_resource

void *last_deallocated_address() const noexcept;

32 Returns: the pointer to the last memory block successfully deallocated by this test_resource

size_t last_deallocated_bytes() const noexcept;

33 Returns: the requested number of bytes of the last memory block successfully deallocated by this test_resource

size_t last_deallocated_alignment() const noexcept;

34 Returns: the requested alignment of the last memory block successfully deallocated by this test_resource
long long allocations() const noexcept;

35 Returns: the total number of allocations requested from this test_resource  [Note: This number includes failed allocations. – End note]

long long deallocations() const noexcept;

36 Returns: the number of total deallocations requested from this test_resource  [Note: This number includes failed deallocations. – End Note]

long long blocks_in_use() const noexcept;

37 Returns: the number of memory blocks still allocated by this test_resource

long long max_blocks() const noexcept;

38 Returns: the largest number of memory blocks allocated at any given time by this test_resource

long long total_blocks() const noexcept;

39 Returns: the total number of memory blocks ever successfully allocated by this test_resource

long long bounds_errors() const noexcept;

40 Returns: the number of buffer overruns and underruns detected by this test_resource.

long long bad_deallocate_params() const noexcept;

41 Returns: the number of mismatched deallocation size and alignment parameters detected by this test_resource

long long mismatches() const noexcept;

42 Returns: the number of mismatched deallocations detected by this test_resource  [Note: Mismatched deallocations are deallocation attempts of memory blocks not obtained from this test_resource. – End Note]

long long bytes_in_use() const noexcept;

43 Returns: the number of bytes currently allocated by this test_resource

long long max_bytes() const noexcept;

44 Returns: the largest number of bytes allocated at any given time by this test_resource

long long total_bytes() const noexcept;

45 Returns: the total number of bytes ever successfully allocated by this test_resource

void print() const noexcept;

46 Effects: Print a report to the standard output that contains the name of this test allocator (if not empty) and describes the current state of this test_resource.  [Note: The printout is intended for human consumption by someone debugging a program. – End Note]

bool has_errors() const noexcept;

47 Returns: false if mismatches() and bounds_errors() and bad_deallocate_params() all return zero and true otherwise
bool has_allocations() const noexcept;

48 Returns: true if blocks_in_use() or bytes_in_use() are non-zero and false otherwise [Note: if either is non-zero both are non-zero. – End Note]

long long status() const noexcept;

49 Returns: 0 if this test_resource has detected no errors and it does not currently have any active allocations (no memory leaks). The number of detected errors if there are any. -1 if there are active allocations (but no errors).

19.12.8.2 Class test_resource_exception

namespace std::pmr {
   class test_resource_exception : public bad_alloc {
   public:
      test_resource_exception(test_resource *originating, size_t bytes, size_t align) noexcept;
      const char *what() const noexcept override;
      test_resource *originating_resource() const noexcept;
      size_t bytes() const noexcept;
      size_t alignment() const noexcept;
   };
}

test_resource_exception(size_t bytes, size_t align) noexcept;

1 Postconditions: bytes() == bytes && alignment() == align

cost char *what() const noexcept override;

2 Returns: an implementation-defined NTBS.

test_resource *originating_resource() const noexcept;

3 Returns: the originating resource pointer supplied at construction

size_t bytes() const noexcept;

4 Returns: the bytes supplied at construction

size_t alignment() const noexcept;

5 Returns: the alignment supplied at construction
### 19.12.8.3 Class `test_resource_monitor`

```cpp
namespace std::pmr {
    class test_resource_monitor {
        public:
            explicit test_resource_monitor(const test_resource& monitored) noexcept;
            explicit test_resource_monitor(test_resource&&) = delete;
            test_resource_monitor(const test_resource_monitor&) = delete;
            test_resource_monitor& operator=(const test_resource_monitor&) = delete;
            void reset() noexcept;
            bool is_in_use_down() const noexcept;
            bool is_in_use_same() const noexcept;
            bool is_in_use_up() const noexcept;
            bool is_max_same() const noexcept;
            bool is_max_up() const noexcept;
            bool is_total_same() const noexcept;
            bool is_total_up() const noexcept;
            long long in_use_change() const noexcept;
            long long max_change() const noexcept;
            long long total_change() const noexcept;
        private:
            long long initial_in_use;  // exposition only
            long long initial_max;     // exposition only
            long long initial_total;   // exposition only
            const test_resource& monitored_resource;  // exposition only
    };

    explicit test_resource_monitor(const test_resource& monitored) noexcept;

1 Postconditions:

- &monitored_resource == &monitored
- initial_in_use == monitored.blocks_in_use()
- initial_max == monitored.max_blocks()
- initial_total == monitored.total_blocks()

void reset() noexcept;

2 Postconditions:

- initial_in_use == monitored_resource.blocks_in_use()
- initial_max == monitored_resource.max_blocks()
- initial_total == monitored_resource.total_blocks()

bool is_in_use_down() const noexcept;

3 Returns: monitored_resource->blocks_in_use() < initial_in_use

bool is_in_use_same() const noexcept;

4 Returns: monitored_resource->blocks_in_use() == initial_in_use

bool is_in_use_up() const noexcept;

5 Returns: monitored_resource->blocks_in_use() > initial_in_use
```
bool is_max_same() const noexcept;

6 Returns: monitored_resource->blocks_max() < initial_max

bool is_max_up() const noexcept;

7 Returns: monitored_resource->blocks_max() > initial_max

bool is_total_same() const noexcept;

8 Returns: monitored_resource->blocks_total() < initial_total

bool is_total_up() const noexcept;

9 Returns: monitored_resource->blocks_total() > initial_total

long long in_use_change() const noexcept;

10 Returns: monitored_resource->blocks_in_use() - initial_in_use

long long max_change() const noexcept;

11 Returns: monitored_resource->blocks_max() - initial_max

long long total_change() const noexcept;

12 Returns: monitored_resource->blocks_total() - initial_total

19.12.8.4 Function template test_resource_loop

namespace std::pmr {
  template <class Block>
  void exception_test_loop(test_resource& tr, Block code);
}

1 Requires: The code argument must be a function object callable with a single test_resource& parameter.

2 Effects: As if by

  for (long long counter = 0; true; ++counter) {
    try {
      tr.set_allocation_limit(counter);
      code(tr);
      tr.set_allocation_limit(-1);
      return;
    } catch (const test_resource_exception& e) {
      if (e.originating_resource() != &tr) {
        reportUnexpectedException(e);
        throw;
      } else if (tr.is_verbose()) {
        reportProgress(counter, tr, e);
      }
    }
  }

3 [Note: The function might never return if code throws a test_resource_exception with the originating resource set to tr other than by calling tr.allocate. — End Note]
ACKNOWLEDGEMENTS

Thanks to John Lakos who has provided the original inspiration, implementation, and much encouragement for this proposal.

Thanks to the colleagues at Bloomberg LP who reviewed the many drafts of this proposal!

REFERENCES


Note that the exception-testing algorithm used in this proposal was developed independently by John Lakos at Bloomberg LP (c. 2002).

https://www.boost.org/community/exception_safety.html

Pablo Halpern, Dietmar Kühl (2018). P0339R4 polymorphic_allocator<> as a vocabulary type