National Body Comments

ISO/IEC PDTS 19568

Technical Specification: C++ Extensions for Library Fundamentals


Document numbers referenced in the ballot comments are WG21 documents unless otherwise stated.
<table>
<thead>
<tr>
<th>MB/NC</th>
<th>Line number</th>
<th>Clause/Subclause</th>
<th>Type of comment</th>
<th>Comments</th>
<th>Proposed change</th>
<th>Observations of the secretariat</th>
</tr>
</thead>
</table>
| JP 1  | 3.2.1       |                  | te             | Current design of apply cannot be used with standard algorithms. This is not consistent with orthogonality policy of C++. We propose make_apply function to make a function object applicable to apply function. For reference, there is a similar design in Boost Fusion Library, fused and make_fused(). This experimental study should be taken into account. | Introduce make_apply as below:  
#include <tuple>  
#include <utility>  

template<typename F, typename Tuple, size_t... I>  
auto apply_impl(F&& f, Tuple&& args,  
std::index_sequence<I...>)  
{  
  return  
  std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);  
}  

template<typename F, typename Tuple, typename Indices =  
std::make_index_sequence<std::tuple_size<Tuple>::value>>  
auto apply(F&& f, Tuple&& args)  
{  
  return apply_impl(std::forward<F>(f),  
  std::forward<Tuple>(args), Indices());  
}  

template<typename F, typename Tuple, size_t... I>  
auto apply_impl(F&& f, const Tuple& args,  
std::index_sequence<I...>)  
{  
  return std::forward<F>(f)(std::get<I>(args))...);  
}  

template<typename F, typename Tuple, typename Indices =  
std::make_index_sequence<std::tuple_size<Tuple>::value>>  
auto apply(F&& f, const Tuple& args)  
{  
  return apply_impl(std::forward<F>(f),  
  std::forward<Tuple>(args), Indices());  
}  

| Note: MB = Member body / NC = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)  
2 Type of comment: ge = general te = technical ed = editorial |
```
::value>>
auto apply(F&& f, const Tuple& args)
{
    return apply_impl(std::forward<F>(f), args, Indices());
}

template <typename F>
class apply_functor {
    F f_;
    public:
        explicit apply_functor(F&& f)
        : f_(std::forward<F>(f)) {}

        template <typename Tuple>
    auto operator()(Tuple&& args)
    {
        return apply(std::forward<F>(f_), std::forward<Tuple>(args));
    }

        template <typename Tuple>
    auto operator()(const Tuple& args)
    {
        return apply(std::forward<F>(f_), args);
    }
};

template <typename F>
apply_functor<F> make_apply(F&& f)
{
    return apply_functor<F>(std::forward<F>(f));
}
```
<table>
<thead>
<tr>
<th>MB/NC</th>
<th>Line number</th>
<th>Clause/Subclause</th>
<th>Type of comment</th>
<th>Comments</th>
<th>Proposed change</th>
<th>Observations of the secretariat</th>
</tr>
</thead>
<tbody>
<tr>
<td>GB 1</td>
<td>6.3.1</td>
<td>p15</td>
<td>Te</td>
<td>The allocator-extended copy constructor for <code>std::experimental::any</code> cannot be implemented as specified, so should be removed. Without this constructor, the value of allocator support in <code>std::experimental::any</code> is questionable.</td>
<td>Suggest removing all constructors taking <code>allocator_arg_t</code> from <code>std::experimental::any</code>.</td>
<td></td>
</tr>
<tr>
<td>GB 2</td>
<td>11.2</td>
<td></td>
<td>Te</td>
<td>Conversion should be provided from/to any specific endianness</td>
<td>Addition of further conversion functions to support conversion to and from big-endian and little-endian representations (as a minimum)</td>
<td></td>
</tr>
</tbody>
</table>

**Usage example:**
```c++
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

int main()
{
    std::vector<std::tuple<int, char, std::string>> v = {
        {1, 'a', "Alice"},
        {2, 'b', "Bob"},
        {3, 'c', "Carol"}
    };

    std::for_each(v.begin(), v.end(),
        make_apply([](int a, char b, const std::string& c)
        {
            std::cout << a << ' ' << b << ' ' << c << std::endl;
        }));
}
```
<table>
<thead>
<tr>
<th>MB/NC</th>
<th>Line number</th>
<th>Clause/Subclause</th>
<th>Paragraph/Figure/Table</th>
<th>Type of comment</th>
<th>Comments</th>
<th>Proposed change</th>
<th>Observations of the secretariat</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI 2</td>
<td>15</td>
<td>[any.cons]</td>
<td>te</td>
<td>Implementation vendors report that the signatures that take an any&amp;&amp; or const any&amp; are unimplementable as currently specified.</td>
<td>Either remove allocator support from any or make it use a polymorphic memory resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI 5</td>
<td></td>
<td>[header.net.synop]</td>
<td>te</td>
<td>As explained in N4249, using the same names for the network byte order conversion functions as the existing posix facilities that may be macros is highly problematic.</td>
<td>Rename the functions so that they do not clash with the existing practice.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI 1</td>
<td>11, 20</td>
<td>[optional.object.observe]</td>
<td>te</td>
<td>As per <a href="https://issues.isocpp.org/show_bug.cgi?id=45">https://issues.isocpp.org/show_bug.cgi?id=45</a>, the rvalue-reference-qualified observers of optional should not return a value, but an rvalue reference instead, in order to ease perfect forwarding and to not cause double-move on emplace to containers. Such a double-move may end up being a double-copy on optionals of legacy types.</td>
<td>Change the signatures to return T&amp;&amp; instead of T and const T&amp;&amp; instead of T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI 4</td>
<td>19</td>
<td>[string.view.access]</td>
<td>ed</td>
<td>The note is confusing. basic_string::data() returns a pointer to a null-terminated buffer regardless of how and from what the basic_string was constructed. How/when is the buffer returned by string_view::data() not null-terminated when a string_view has been constructed from a literal, and how is it typical that passing data() to a function expecting a null-terminated char* a mistake?</td>
<td>Clarify or strike the note.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FI 3</td>
<td>6</td>
<td>[string.view.cons]</td>
<td>ed</td>
<td>&quot;Constructs a basic_string_view referring to the same string as str,&quot;. str doesn't refer to a string, and the wording is inconsistent with similar constructors for basic_string in the standard proper, where such charT* are said to &quot;point to an array&quot;. See [string.cons] for reference.</td>
<td>Use the same terminology as the standard basic_string specification uses.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>